Improving Efficiency of Parallel Vertex-Centric Algorithms for Irregular Graphs

Muhammet Mustafa Ozdal, *Member, IEEE*

Abstract—Memory access is known to be the main bottleneck for shared-memory parallel graph applications especially for large and irregular graphs. Propagation blocking (PB) idea was proposed recently to improve the parallel performance of PageRank and sparse matrix and vector multiplication operations. The idea is based on separating parallel computation into two phases, *binning* and *accumulation*, such that random memory accesses are replaced with contiguous accesses. In this paper, we propose an algorithm that allows execution of these two phases concurrently. We propose several improvements to increase parallel throughput, reduce memory overhead, and improve work efficiency. Our experimental results show that our proposed algorithms improve shared-memory parallel throughput by a factor of up to 2x compared to the original PB algorithms. We also show that the memory overhead can be reduced significantly (from 170 percent down to less than 5 percent) without significant degradation of performance. Finally, we demonstrate that our concurrent execution model allows asynchronous parallel execution, leading to significant work efficiency in addition to throughput improvements.

Index Terms—Parallel algorithms, graph algorithms, sparse matrix vector multiplication (SpMV), sparse matrix sparse vector multiplication (SpMSpV), high performance computing

1 Introduction

In vertex-centric programming model, data accessed by vertex v is typically localized to the data of v plus its immediate neighbors. Such algorithms can also be specified using sparse matrix and dense/sparse vector multiplication (SpMV/SpMSpV) operations. It was shown that many graph algorithms can be modeled using these basic operations [1].

Distributed graph frameworks are scalable and can handle massive amounts of data. On the other hand, shared-memory systems can lead to more efficient parallelism due to reduced communication costs between threads. Furthermore, current server-grade systems have large enough DRAM memories to handle graphs with tens of billions of edges.

In the context of shared-memory parallelism, graph applications are known to be memory-access bound due to relatively low amount of computation performed per data brought from memory. Many real-world graphs are known to be irregular and scale-free, where the vertex degree distribution follows power law [2]. Such graphs are hard to partition, and existing reordering and cache blocking based methods have been shown to be ineffective for them [3]. As a result, low temporal and spatial locality inherent in irregular graphs lead to inefficient utilization of the available memory bandwidth.

Recently, a technique called Propagation Blocking (PB) was proposed to improve the performance of shared-memory parallel PageRank [3] and SpMV [4] kernel operations. The idea is based on separating vertex-centric computation into two phases, *binning* and *accumulation* such that random accesses to DRAM are replaced with contiguous accesses. This idea was later extended to the SpMSpV kernel for work efficient execution [5]. These works have reported significant parallel performance improvements especially for large and irregular graphs.

Despite performance improvements, the PB algorithm requires significant (up to ~200%) extra memory to store the intermediate data between binning and accumulation phases. Increasing the system memory requirements by up to ~3x compared to a compressed sparse row/column (CSR/CSC) graph restricts the maximum graph sizes that can be handled by a single node, limiting the advantages of shared-memory parallelism.

In this paper, we propose several improvements over the original PB algorithm for large and scale-free graphs. Specifically, our improvements are applicable to vertex-centric graph algorithms that can also be represented as SpMV and/or SpMSpV operations. Our algorithms increase throughput of computation, reduce memory overhead, and improve work efficiency. The contributions of this paper can be summarized as follows:

- We propose a parallel execution model that allows concurrent execution of the binning and accumulation threads (Section 3). This model allows storing only a small fraction of the intermediate data, and hence reduces the extra memory requirements without significant performance degradation.
- We propose an optimized graph layout for iterative graph algorithms (Section 4). This layout improves throughput of computation using only 4-bit local

• *The author is with the Computer Engineering Department, Bilkent University, Ankara 06800, Turkey. E-mail: mustafa.ozdal@cs.bilkent.edu.tr.*

Manuscript received 27 Nov. 2018; revised 11 Mar. 2019; accepted 14 Mar. 2019. Date of publication 19 Mar. 2019; date of current version 11 Sept. 2019. (Corresponding author: Muhammet Mustafa Ozdal.)
Recommended for acceptance by K. Madduri.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2019.2906166

1045-9219 © 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

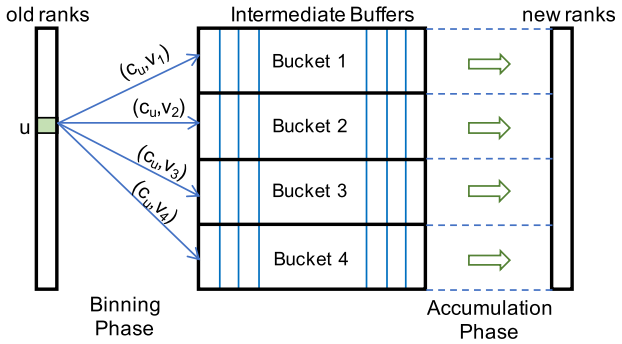

Fig. 1. Two-phase propagation blocking algorithm illustrated for four buckets. The binning phase is shown for a vertex u with four out-neighbors $v_1 - v_4$, which belong to buckets $1 - 4$, respectively.

buffer pointers. It also leads to efficient storage and memory transfers of vertex indices through simple packing mechanisms.

- For scale-free graphs, we propose a two-level bucketing technique that allows better performance tradeoffs between binning and accumulation threads (Section 5).
- We propose an analytical model to estimate the best set of bucketing parameters that will lead to highest performance. We show that these models have high fidelity and can be used for automatic parameter selection (Section 6).
- We show that our proposed concurrent execution model allows asynchronous parallel execution, which reduces staleness of data. We extend our framework for work efficient execution through the use of delta caching idea [6] and active vertex set data structure (Section 7).

Our experiments on multiple graph applications show that our algorithms lead to up to 4.2x parallel throughput improvements over traditional pull-based implementations, while the original PB algorithm achieves up to 2.5x improvement. Furthermore, these improvements are achieved while substantially reducing the memory overheads. For work efficient execution, we show that our concurrent execution model helps reduce the total amount of work done in addition to throughput improvements, leading to overall parallel speedup of up to 3.3x compared to the baseline PB algorithm.

2 Background

In this section, we use PageRank [7] as a running example, because it is a commonly used representative benchmark that captures the common execution patterns of many graph applications [8], [9], [10], [11]. PageRank was originally proposed for search engines to measure the relative importance (i.e., *rank*) of web pages based on the topology of a web graph $\mathcal{G}$. Let V and E denote the set of vertices and edges in $\mathcal{G}$, respectively. The rank of vertex v (denoted as p_v) is defined as follows:

$$p_v = \frac{1-\beta}{|V|} + \beta \sum_{(u \to v) \in E} \frac{p_u}{d_u}. \quad (1)$$

Here, d_u denotes the out-degree of vertex u and β is a constant parameter that determines the teleportation probability of the random walk. In one iteration of the PageRank algorithm, all p_v values are computed using this formula. Multiple iterations are performed until convergence.

It is well known that PageRank computation kernel is a special case of sparse matrix vector multiplication (SpMV) operation [1]. The algorithms described in the rest of the paper can also be adapted for SpMV.

For a graph stored in a vertex-centric format, such as compressed sparse row (CSR) or compressed sparse column (CSC), the traditional implementations commonly use either pull or push patterns to compute PageRank [12]. In both patterns, an outer loop iterates over each vertex v and an inner loop iterates over the neighbors of the current vertex v. For large and unstructured graphs, access to the data of a neighbor typically requires a random access to DRAM due to poor temporal and spatial locality. In other words, most of the data accesses in the inner loop are expected to require access to main memory, and they become the main performance bottleneck. Further details of the pull and push execution patterns are provided in *Section 1* of the *Supplemental Material*, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2019.2906166.

For unstructured graphs, the available memory bandwidth is not utilized efficiently by the inner loops of pull and push based algorithms due to low spatial locality [13]. As an example, let us assume that each page rank value is 4 bytes and the cache line size is 64 bytes (as in today's mainstream x86 architectures). Each access to neighbor data in the inner loop potentially requires transfer of 64 bytes from/to DRAM, out of which only 4 bytes contain useful data. In the following section, we summarize a recently proposed algorithm that improves both access locality and memory bandwidth utilization for unstructured graphs.

2.1 Propagation Blocking

The propagation blocking algorithm was recently proposed by two research groups concurrently [3], [4] to alleviate the random memory access bottlenecks described in the previous section. The pseudo-code is listed in Algorithm 1 and an example is illustrated in Fig. 1. For simplicity of presentation, two arrays (old and new) are used to store the vertex data values. However, an implementation with only a single array is also possible.

In this algorithm, K buckets are defined over the set of vertices V such that each vertex v belongs to exactly one bucket. Let $Bin(v)$ be a function that maps vertex v to its defined bucket.[1] The algorithm consists of two distinct phases: *binning* and *accumulation*.

The outer loop of the binning phase proceeds similar to a push-based algorithm, where the contribution of the current vertex u is computed as $c_u = \beta p_u / d_u$. However, instead of adding c_u to the rank of each out-neighbor immediately, c_u is written to an intermediate buffer corresponding to its bucket. In the example of Fig. 1, vertex u has four out-neighbors $v_1 - v_4$, which belong to buckets $B_1 - B_4$, respectively. So, a tuple (c_u, v_i) is added to the buffer corresponding to B_i

1. For runtime efficiency, a simple assignment function, such as shifting the vertex index to right by a certain amount, is used to determine the bucket of v.

for $1 \leq i \leq 4$. This operation is performed for all vertices u in V, and this is defined as the binning phase of the algorithm. In the *accumulation* phase, each bucket is processed independently. For bucket B, the tuples in the corresponding buffer are processed and the c_u values are added to the new ranks of the corresponding destination vertices.

Algorithm 1. PageRank Using Propagation Blocking [3]

Input: $\mathcal{G} = (V, E)$
1: Initialize $oldRanks$ array with initial ranks
2: **repeat**
3: // *Binning Phase*:
4: **for** each vertex u in V **do**
5: $c_u = \beta \times oldRanks[u]/d_u$ // *contribution of vertex u*
6: **for** each out-neighbor v_i of u **do**
7: add (c_u, v_i) to the buffer of $Bin(v_i)$
8: **end for**
9: **end for**
10:
11: // *Accumulation Phase:*
12: **for** each bucket B **do**
13: **for** each v that belongs to B **do**
14: $newRanks[v] = (1 - \beta)/|V|$ // *initialize ranks*
15: **end for**
16: **for** each (c_u, v) in buffer corresponding to B **do**
17: $newRanks[v] \mathrel{+}= c_u$ // *add contribution of vtx u*
18: **end for**
19: **end for**
20: $oldRanks \leftarrow newRanks$
21: **until** convergence condition satisfied

This algorithm can be parallelized in a straightforward way. In the binning phase, each thread can be assigned a set of vertices and a unique buffer space to make sure that different threads write to different memory locations. Then, a global barrier is needed before the accumulation phase begins. In the accumulation phase, each thread is assigned a set of buckets. Since a vertex belongs to exactly one bucket and each bucket is processed by a single thread, it is guaranteed that there will be no race conditions as the new rank values are computed.

The main advantage of this algorithm is improved memory access locality for large and irregular graphs. In the binning phase, each thread reads the old ranks of the source vertices in consecutive order, leading to high spatial locality (line 5 of Algorithm 1). Similarly, each thread writes consecutively to K different buffers (line 7). If K, the number of buckets, is chosen small enough such that K cache lines can fit into the L1 or L2 cache of one core, then these write operations are expected to have high spatial locality.

In the accumulation phase, one bucket B is processed at a time. If K is chosen large enough (i.e., the number of vertices per bucket is small enough), then the vertex data corresponding to the vertices that belong to B can fit into a local cache. This leads to high temporal locality for accesses to the vertex data (lines 14 and 17). On the other hand, the tuples from the intermediate buffer are read in consecutive order (line 16), leading to high spatial locality.

Note that K must be chosen considering the tradeoff between the binning and accumulation phases. It has been shown that as K is increased, the performance of binning gets worse while the performance of accumulation improves, and vice versa.

For iterative graph algorithms that process all vertices in every iteration, a deterministic layout was proposed by [3] and [4] for the destination indices in the intermediate buffers. More specifically, the contribution (c_u) and the destination index (v_i) values are stored separately in two different data structures. Since the destination indices do not change across iterations, they are written only once in the beginning. During the iterations, only the c_u values change and they need to be written in the intermediate buffers (line 7 of Algorithm 1). This reduces the number of bytes written to DRAM during the binning phase and improves performance.

Different heuristics were proposed by [3] and [4] to implement the inner loop of the binning phase efficiently. In line 7 of Algorithm 1, a tuple needs to be written to the next available location of the buffer corresponding to $Bin(v_i)$. One can maintain K counters, one for each bucket, to determine these locations on the fly. Buono et al. [4] proposed to precompute the exact buffer location that needs to be updated for each inner loop iteration and store it in the graph. This corresponds to storing a write pointer corresponding to each edge. This heuristic saves some computation time, but increases the graph size (extra 4 bytes per edge) and the amount of data transferred from DRAM. On the other hand, Beamer et al. [3] propose a different heuristic to reduce the runtime of the binning phase. A small local buffer is defined for each bucket, and the writes (line 7 of Algorithm 1) are first done into these local buffers. Once a local buffer is full, it is flushed to the global memory using vectorized streaming store instructions.

An important downside of the propagation blocking algorithm is the large extra memory overhead due to the intermediate buffers. For each edge of the graph, a tuple of source vertex contribution and destination vertex index needs to be stored in memory. We will show in Section 4 that this may lead to up to 200 percent extra memory overhead. For execution on a single node, increased DRAM capacity is needed to handle up to 3x larger memory requirements, limiting the maximum size graphs that can be handled by a single node. In the next section, we propose an execution model that can reduce the memory overhead due to intermediate buffers substantially without sacrificing performance too much.

3 Concurrent Execution Model

We have performed performance bottleneck analysis for the original pull-based and the PB-based PageRank implementations using the top-down methodology [14] available in Intel's Vtune software. Our results on large and scale-free graphs have shown that the main performance bottleneck of the original algorithm is DRAM read access, with stall cycles of up to 80 percent. However, we have observed that DRAM read access is no longer the main performance bottleneck for the PB-based PageRank implementation. (Details of this analysis are provided in *Section 2* of the *Supplemental Material*, available online.) However, this improvement in DRAM access efficiency comes at the expense of increased memory usage.

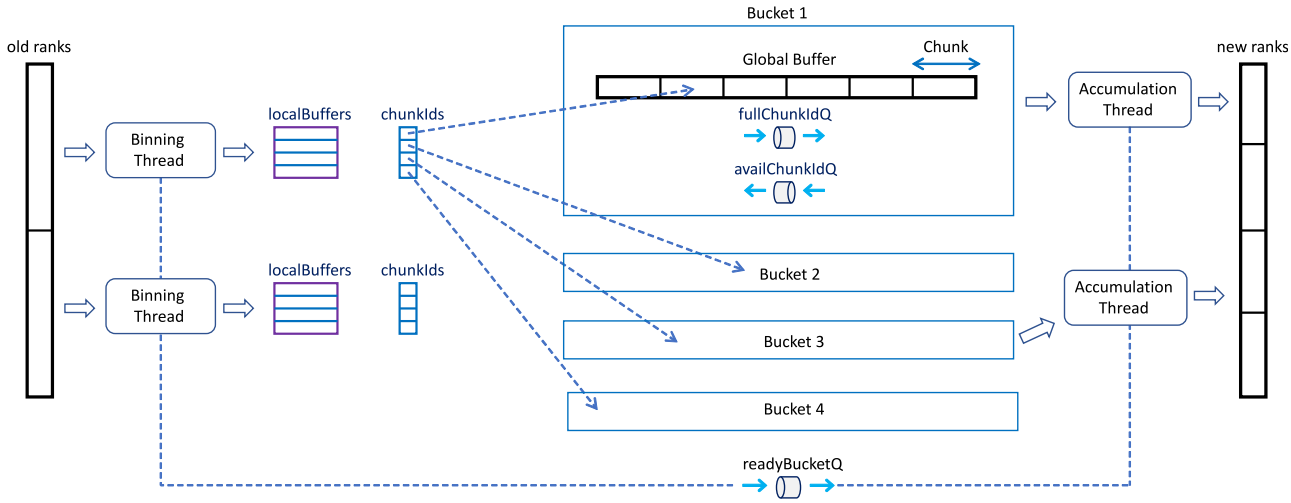

Fig. 2. Concurrent propagation blocking algorithm illustrated for four buckets.

Our idea is to introduce a tradeoff between memory overhead and the number of DRAM accesses. Instead of performing accumulation after all the tuples are generated as in Algorithm 1 (referred to as the *two-phase* algorithm from now on), we propose an execution model where binning and accumulation operations are done concurrently. This allows reducing the size of the intermediate buffers significantly. However, if accumulation is done multiple times within an iteration for a subset of vertices, the corresponding vertex data needs to be brought from DRAM to local cache multiple times. This is expected to increase the number of DRAM accesses, but our aforementioned performance analysis shows that DRAM accesses are no longer a significant bottleneck in the PB algorithm, and there is some room for tradeoff. Our experiments in Section 9 will show that this intuition is indeed valid in practice.

Algorithm 2. Binning Thread for Concurrent Execution

Input: (V_T, E_T) // *subgraph assigned to this thread*
1: **for** each bucket B_i **do**
2: $chunkIds[i] \leftarrow availChunkIdQ[i]$.dequeue()
3: **end for**
4: **for** each vertex u in V_T **do**
5: $c_u \leftarrow \beta \times oldRanks[u]/d_u$ // *contribution of vertex* u
6: **for** each out-neighbor v_i of u **do**
7: $k \leftarrow Bin(v_i)$ // *destination bucket*
8: add (c_u, v_i) to $localBuffer[k]$
9: **if** $localBuffer[k]$ is full **then**
10: $C \leftarrow$ chunk $chunksIds[k]$ of global buffer k
11: flush $localBuffer[k]$ to the end of C
12: **if** C is full **then**
13: $fullChunkIdQ[k]$.enqueue(chunkIds[k])
14: **if** $availChunkIdQ[k]$ is *almost* empty and bucket k is not locked **then**
15: $readyBucketQ$.enqueue(k)
16: **end if**
17: $chunkIds[k] \leftarrow availChunkIdQ[k]$.dequeue()
18: **end if**
19: **end if**
20: **end for**
21: **end for**

3.1 Concurrent Algorithm

Our proposed concurrent execution model is illustrated in Fig. 2, where the binning and accumulation threads can be considered as producers and consumers, respectively. The tasks generated by the producers are passed to consumers through the task queue *readyBucketQ*, which stores the indices of the buckets that are ready for accumulation. For efficient data communication between producers and consumers, the global buffers are divided into smaller chunks. These chunks are managed for each bucket independently. The empty chunk indices are stored in queue *availChunkIdQ* and are utilized by the binning threads. Once a chunk is completely populated by a binning thread, its index is enqueued into *fullChunkIdQ* and is eventually dequeued by an accumulation thread. After its contents are consumed by an accumulation thread, it is enqueued again to *availChunkIdQ*. In other words, the limited intermediate buffer space is continuously recycled between producers and consumers to reduce the required memory size significantly. Note that the queues shown in Fig. 2 are implemented in a thread-safe manner by using atomic instructions and simple spin locks to guarantee race free execution. Furthermore, their sizes are chosen large enough to guarantee that there are no deadlocks.

The pseudo code for a binning thread is listed in Algorithm 2. In the beginning (lines 1-3), one chunk is allocated for each bucket i and is stored in $chunkIds[i]$. As in the two-phase algorithm, each binning thread processes the vertices assigned to it to generate tuples of form (c_u, v_i), where c_u is the contribution of the source vertex and v_i is the index of the destination vertex. Each tuple is written to the local buffer corresponding to the bucket of the destination vertex (line 8). It is assumed that local buffers are small enough to fit into the L2 cache of a binning thread. Similar to the optimization proposed in [3], these local buffers are used to coalesce individual writes so that global memory store operations are done in cache line granularity.

Once a local buffer is full (line 9), it is flushed to the global buffer using vectorized streaming write instructions (line 11). The location in the global buffer is determined by the chunk C allocated for the corresponding bucket. When all the space for the current chunk is utilized (line 12), it is enqueued to

fullChunkIdQ (line 13) so that an accumulation thread can consume its data. Once there are enough number of chunks populated by the binning threads (line 14), a new task is created corresponding to the current bucket and enqueued to *readyBucketQ* (line 15). The execution of the binning thread continues after it allocates a new chunk (line 17).

The pseudo code for the accumulation thread is listed in Algorithm 3. The accumulation threads receive the tasks defined by the binning threads through *readyBucketQ* (line 2), which stores the bucket indices that have sufficient number of chunks to process. Once an accumulation thread starts to process a bucket k, it is guaranteed that no other accumulation thread can process it. This is ensured through a locking mechanism (line 3 in Algorithm 3) and making sure that there is at most one copy of each bucket in *fullChunkIdQ* (line 14 in Algorithm 2). For simplicity, the details of the locking mechanism are not included in the pseudo codes provided. Once the accumulation thread starts processing bucket k, it processes all chunks stored in the corresponding *fullChunkIdQ* (line 5 of Algorithm 3). Once all the tuples in a chunk are processed (lines 6-8), it is put back to the corresponding *availChunkIdQ* (line 9) to allow reuse by the binning threads. After all its available chunks are processed, bucket k is unlocked (line 11). The execution continues until all binning threads finish processing their subset of vertices and the task queue, *readyBucketQ*, is empty.

For brevity, the given pseudo codes do not provide low level implementation details such as how to handle partial chunks at the end of an iteration, etc. However, important design choices are discussed in the next section.

3.2 Design Choices

The first design choice is how to determine the mix of the binning and accumulation threads while ensuring that all cores in the system are effectively utilized. In our implementation, we make this determination dynamically based on the available tasks. In the beginning of an iteration, the number of binning threads is set to be equal to the number of cores and there are no accumulation threads. As the chunks of the global buffers are populated over time, the number of available chunks to be written by the binning threads decrease. If a binning thread cannot dequeue a chunk index from *availChunkIdQ* (line 17 of Algorithm 2), it switches to become an accumulation thread[2] to execute one iteration of the loop in Algorithm 3. A similar switch occurs when a binning thread finishes processing the subgraph assigned to it. This design allows achieving load balance between binning and accumulation threads in a simple and effective way.

Another design choice is how to determine the sizes of the local and global buffers. As mentioned before, local buffers are used to perform DRAM writes in cache line granularity. Hence, the size of each local buffer can be chosen to be one cache line (64 bytes for a Xeon system). The number of chunks in a global buffer (corresponding to one bucket) is determined based on the number of binning threads. As explained in Algorithm 2, each binning thread has one chunk assigned to it per bucket. We have observed in our experiments that choosing the number of chunks to be larger than twice the number of threads is sufficient.[3] Since the number of chunks is determined based on the number of threads, we can control the total size of the intermediate buffers by changing the size of one chunk. We will show that a tradeoff between performance and memory size can be achieved by changing this parameter.

Algorithm 3. Accumulation Thread for Concurrent Execution

```
1: repeat
2:   k ← readyBucketQ.dequeue()
3:   lock bucket k for accumulation
4:   while fullChunkIdQ[k] is not empty do
5:     C ← fullChunkIdQ[k].dequeue()
6:     for each (c_u, v) in C do
7:       newRanks[v] += c_u
8:     end for
9:     availChunkIdQ[k].enqueue(C)
10:  end while
11:  unlock bucket k
12: until no active binning thread and empty readyBucketQ
```

Similar to the previous algorithms [3], [4], the number of buckets is an important parameter that determines the performance tradeoff between binning and accumulation operations. We propose an optimization about how to choose this parameter in Section 5, together with an analytical model that allows choosing this parameter automatically based on the characteristics of the platform and the input graph (Section 6).

3.3 Discussion

Note that the memory management operations in the inner loop of Algorithm 2 (lines 9-19) are amortized over multiple iterations. In our implementation, each local buffer can hold 16 elements, so one flush-to-global-buffer operation is amortized over 16 iterations. Similarly, each chunk is defined to have 2,048 elements or more,[4] so the queue operations (lines 13-17 of Algorithm 2 and lines 5,9 of Algorithm 3) are amortized over at least 2,048 iterations. Compared to the two-phase PB algorithm, our proposed algorithm requires explicit management of the intermediate buffers. However, since these management operations are amortized over many iterations, their impact on performance is expected to be negligible.

On the other hand, there is a potential degradation of accumulation thread performance due to increased cache misses. As explained earlier, it is expected that the random accesses to newRanks subarray (line 7 of Algorithm 3) have high temporal and spatial locality. When an accumulation thread is assigned a bucket k (line 2 of Algorithm 3), it processes all the chunks that belong to k (lines 4-10 of Algorithm 3). If the number of entries in the intermediate buffer is as large as the number of edges, then each bucket is expected to be processed only once within one iteration (as in the two-phase algorithm). In that case, the newRanks

2. This is actually a conceptual switch performed in the code by the same Posix thread.

3. In our experimental platform, there are 24 cores, and we have set the number of chunks to be equal to 64.

4. Details of parameter selection will be provided in Section 9.

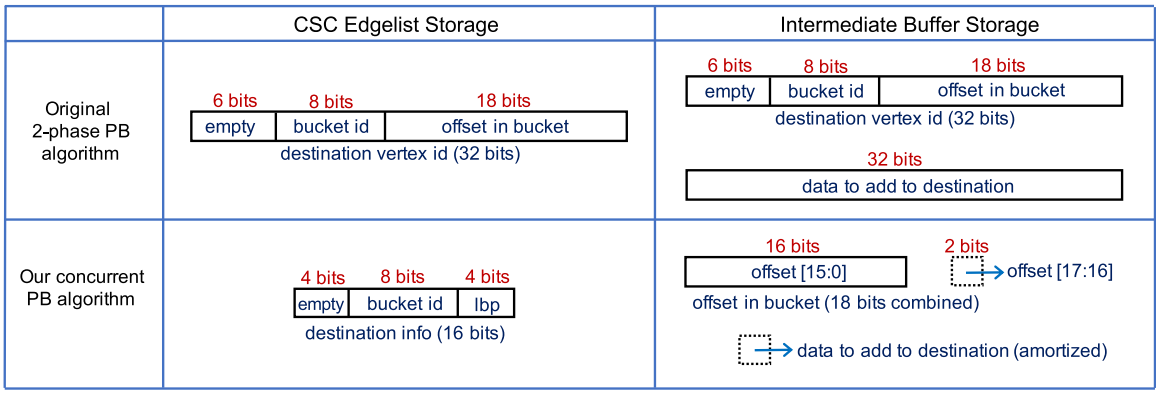

Fig. 3. An example demonstrating memory storage requirements per edge when an unweighted graph has up to 64M vertices and the number of buckets is 256. The additional storage for the PB algorithm is 64 bits per edge compared to a traditional CSC layout (~200% memory overhead for an unweighted graph). On the other hand, the overhead of our proposed layout is only 2 bits per edge plus amortized cost of data storage in intermediate buffers.

entries are expected to be brought from DRAM only once due to cold misses. As the intermediate buffer size is reduced (as in our proposed algorithm), the number of times a bucket k is processed and hence the number of cold misses for the newRanks array increases.

However, these cold misses are still amortized over multiple accesses. For example, assume that there are 32K vertices in bucket k and there are 128K entries in the corresponding chunks to be processed (in lines 4-10 of Algorithm 3). Assuming 64 bytes per cache line and 4 bytes per rank value, there are 2K cache lines that need to be brought from DRAM due to cold misses. These cache lines will be accessed 128K times in the inner loop (line 7), hence each cache line access will be amortized over 64 operations on average. This amortization factor is proportional to the size of the intermediate buffer size.[5] We will show in our experiments that the intermediate buffer size can be reduced significantly compared to the two-phase algorithm with minimal impact to performance.

4 Optimized Graph Layout

In this section, we focus on the storage requirements per graph edge assuming that the total graph storage size is proportional to the number of edges. For the algorithms we discuss, the per-vertex storage is the same as the standard CSR/CSC representations. For simplicity of presentation, we will explain the basic concepts in this section through the help of an example: Assume that our input graph has up to 64 million vertices, and the number of buckets for the PB algorithm is chosen to be 256.

The PB algorithm by Beamer, et al. [3] can be applied directly on an input graph stored in CSC format. In a CSC based graph layout, the destination vertex ID for each edge needs to be stored in the edgelist. For our example, at least 26 bits are needed to represent a vertex ID, hence a 32-bit word is used per edge just to store the layout information. Out of the 26 bits, 8 bits can be interpreted as the bucket ID and the remaining 18 bits can be interpreted as the offset within the bucket. This is shown in the top-left quadrant of Fig. 3.

In the original two-phase PB algorithm, the accumulation operations begin only after all binning is done. Hence, all the tuples (c_u, v) generated in the binning phase (line 7 of Algorithm 1) need to be stored in an intermediate buffer. The tuple stored for one edge is shown in the upper-right quadrant of Fig. 3 for [3]. Note that while the original CSC layout needs only 32 bits per edge for an unweighted graph, an extra 64 bit storage per edge is needed by the PB algorithm. This is expected to increase the total memory size by up to an extra ~200%, compared to the original graph.

The graph layout proposed by Buono, et al. for the PB algorithm [4] is slightly different. The bucket offset is not stored in the CSC edgelist, so the CSC edgelist requires only 2 byte storage per edge. Similarly, the bucket ids are not stored in the intermediate buffer. The authors assume that bucket offsets can be stored in 16-bit words. (Note that this assumption may not hold for very large graphs as in our example, where bucket offsets are 18 bits.) As a result, they have 2-byte indices stored per edge for both the CSC edgelist and for the intermediate buffer. However, an additional 32-bit write address per edge is embedded in the CSC edgelist for efficient binning operations. Combined with the 32-bit data word, the total overhead per edge again becomes 64 bits, the same as in [3].

For iterative algorithms that process all vertices every iteration, the bucket offsets need to be written to the intermediate buffers only once in the beginning. In the consecutive iterations, these offsets do not need to be updated by the binning threads, as explained in Section 2.1. We can incorporate this optimization into our concurrent execution model by storing the destination offsets for all edges in the beginning, similar to the previous algorithms [3], [4]. If this is done in a straightforward way, it will lead to extra 32-bit storage per edge, i.e., ~100% memory overhead with respect to the original graph. We propose improvements to reduce this overhead as explained below.

5. Note that this particular example assumes that there are more than 128K edges connected to these 32K vertices, i.e., the average vertex degree is larger than 4. For this example, if the average vertex degree was less than 4, then the chunk size parameter would need to be decreased accordingly, which would lead to less amortization of cold misses. For graphs with very small average vertex degrees (e.g., 1-2), we expect the benefits of concurrent execution to be limited compared to the original 2-phase PB algorithm, because the number of entries stored per vertex in the intermediate buffers (and hence the amortization factor) would already be small in the original 2-phase PB algorithm.

Our proposed per-edge storage is demonstrated in the lower half of Fig. 3. In the CSC edgelist, we store 16 bits per edge, which consists of 1) the *bucket ID* and 2) the *local buffer write pointer (lbp)*. As explained in Algorithm 2, a local buffer is used per bucket to coalesce writes to the same cache line. Since we assume 4-byte data words and 64-byte cache lines, there are 16 entries per local buffer in our implementation. The 4-bit *local buffer write pointer (lbp)* indicates which location the current edge data should be written to in the local buffer. This is similar to the write-pointer optimization of Buono, et al. [4], but we store only 4 bits per edge instead of 32 bits. In other words, while Buono, et al. store the exact 32-bit index of the intermediate buffer entry corresponding to the current edge, we only store the 4-bit index of the local buffer entry. This allows our binning threads to determine the write index of the corresponding local buffer entry using a simple bitwise mask operation. We still maintain counters to keep track of which location to write to in the global buffers, but flushing from local buffer to global buffer is amortized over 16 write operations. Using 4-bit pointers for local buffers, but maintaining counters for global buffers gives us a good tradeoff point between performance and memory size.

The bucket offsets are stored in the intermediate buffers, and they are consumed by the accumulation threads. If the bucket offset for each edge is less than or equal to 16 bits, then we store the offsets in 16-bit words. Otherwise, we use a simple packing scheme to store these offsets efficiently. For the example of Fig. 3, each bucket offset is 18 bits. In this case, the lower bits are stored in a 16-bit word, whereas the remaining 2 bits are packed into a 16-bit word together with data from 7 other edges. Obviously, such a packed storage scheme brings some compute overhead for the accumulation threads. For this example, the upper 2 bits need to be unpacked and merged with the lower 16 bits to compute the original 18-bit offset. However, we have observed that the reduction in DRAM transfer volume (e.g., 18 bits per edge instead of 32 bits) more than compensates for this small compute overhead.

Observe that the storage overhead of destination indices is only 2 bits per edge for this particular example, with respect to the original CSC representation. Furthermore, for the execution model we propose in Section 3, the intermediate buffer size for data words can be chosen to be much smaller than the number of edges, because the buffers populated by the binning threads are concurrently consumed by the accumulation threads. Hence, the overhead due to data storage in intermediate buffers is amortized over all edges (shown as a dashed box in the lower-right quadrant of Fig. 3). We will show in our experiments that our memory overhead can be reduced to less than 5 percent, while the previous PB algorithms may require close to 200 percent extra memory.

Although we have presented our proposed layout using a specific example, the actual bit counts are determined based on the input graph size and the number of buckets chosen.

Generating the optimized graph layout has a preprocessing overhead that is linear in the number of edges. This conversion can be done on-the-fly as the PB algorithm is executed on the input graph in the first iteration. This graph layout is specifically optimized for PB-style algorithms. In other words, it is not optimal for traditional (e.g., pull or push-based) algorithms, because the *destination vertex id* is split and stored in two different locations, as illustrated in the Fig. 3. This may be a potential limitation if the same graph data structure in memory is to be used for both PB-style and traditional algorithms. Although conversion between the two layouts is expected to be fast, the proposed layout is an optional feature and it is orthogonal to the other improvements proposed in the rest of this paper.

5 Two Level Bucketing

In this section, we propose an optimization that allows a better tradeoff point between binning and accumulation thread performance by using different bucket sizes for large-degree and small-degree vertices.

It has been shown that there is a performance tradeoff between binning and accumulation threads of the original PB algorithm [3], [4]. Let k denote the number of buckets chosen as the algorithm parameter. At a given time, a binning thread can write to k different cache lines. It was observed that the performance of the binning threads degrade as k is increased due to increased L1 and/or L2 cache misses. On the other hand, one accumulation thread accesses the vertex data belonging to one bucket at a given time, hence the number of vertices processed by an accumulation thread is $|V|/k$, where $|V|$ is the number of vertices in the graph. As k is increased, the performance of an accumulation thread is expected to improve due to the decrease in its working set size.

This tradeoff is shown in Fig. 4b for the *friendster* dataset, which has $125M$ vertices and $1.8B$ edges. Here, the x-axis is the number of vertices per bucket, (denoted as *bucket size* and computed as $|V|/k$) and the y-axis is the throughput of computation, defined as the number of edges processed per second. This performance data was obtained by running the proposed concurrent algorithm for PageRank on a two-socket Xeon E5-2650-v4 system with 24 cores. As shown in this figure, the throughput of accumulation operation drops almost linearly with increasing bucket size due to the increased working set size per accumulation thread. On the other hand, the binning performance improves as the bucket size is increased because of the decrease in the number of buckets (k). The net result is almost flat overall performance for a large range of bucket sizes, from 32K to 256K.

We propose an optimization specific to scale-free graphs to improve this performance tradeoff. In a scale-free graph, a small percentage (e.g., 20 percent) of vertices are connected to a large percentage (e.g., 80 percent) of edges, also known as the 80-20 rule. It was shown that many real-world graphs have this property [15]. The edge distribution for the *friendster* graph is shown in Fig. 4a, where the vertical dashed line indicates that 12.5 percent of the largest-degree vertices are connected to the 84 percent of edges.

Our proposed optimization is to define two different bucket sizes for 1) large-degree vertices and 2) small-degree vertices. We will show that this can achieve a better performance tradeoff point between binning and accumulation. The intuition is that large-degree vertices are processed more often than small-degree vertices since they are connected to most of the edges. Hence, a smaller bucket size

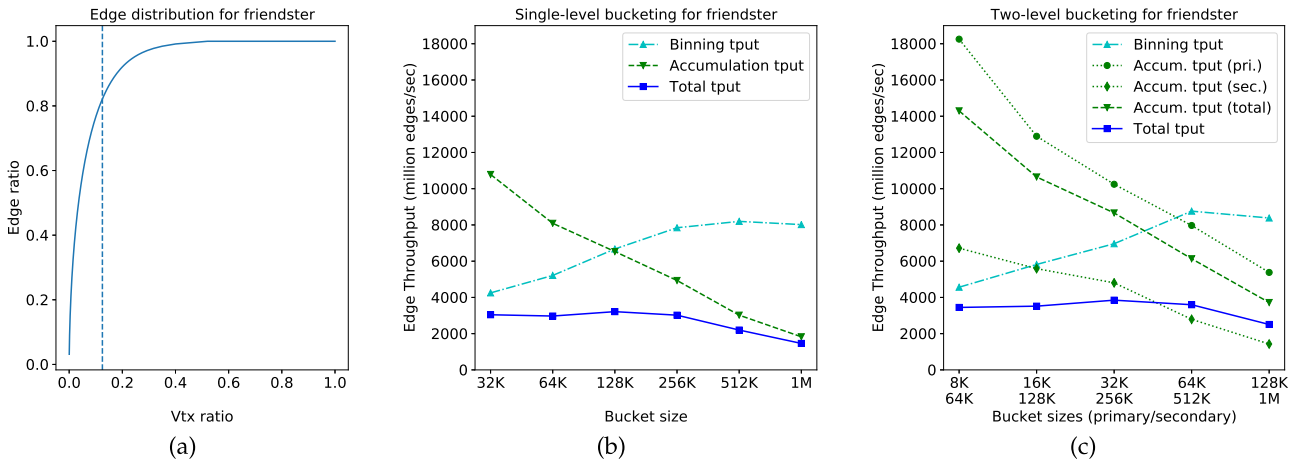

Fig. 4. (a) The vertex degree distribution for the *friendster* graph. The throughput of binning and accumulation operations as a function of bucket size for (b) single-level bucketing and (c) two-level bucketing.

can be chosen for them to improve the performance of accumulation.

For this, we first categorize the vertices in G as *primary* and *secondary* based on the number of edges they are connected to. Let V_P and V_S denote the set of primary and secondary vertices, where the degree of any vertex $v_p \in V_P$ is greater than or equal to the degree of any vertex $v_s \in V_S$. This categorization is done using an input parameter called *primary vertex ratio* (denoted as r_v) such that $|V_P| = r_v \cdot |V|$. For the example of Fig. 4a, if $r_v = 1/8$, then $1/8$th of the vertices (left of the vertical dashed line) are categorized as *primary*, and the rest as *secondary*. Observe that the set of primary vertices in this example are connected to 84 percent of the edges.

Given an input r_v parameter, we can reorder the vertex indices such that the indices of the primary vertices are less than the indices of the secondary vertices. This can be done in linear time during construction of the input graph, using an algorithm such as described in [16].

After the vertices are categorized this way, two different bucket parameters can be chosen. Let k_p and k_s denote the number of buckets for the primary and secondary vertex sets, respectively. Fig. 4c shows an example of how performance changes for the *friendster* graph if r_v is chosen to be $1/8$ and k_p is equal to k_s. As the value of $k_p = k_s$ is decreased, both the primary and secondary bucket sizes increase, hence performance of accumulation threads degrade. However, since the bucket size of the primary vertex set (V_P) is chosen to be much smaller than the secondary vertex set (V_S), the accumulation performance for V_P turns out to be much higher than that of V_S. Since V_P covers more than 80 percent of the edges, the total accumulation performance also improves significantly. Compared to single-level bucketing for the same example (Fig. 4b), the improvement in accumulation performance is achieved without a noticeable degradation in binning performance. Hence, the overall performance is also improved using two-level bucketing - by about 20 percent in this example.

Implementing two-level bucketing requires only a minor change in the binning threads. Given a neighbor vertex index n_v, we first need to determine whether n_v belongs to V_P or V_S. In our implementation, we always choose $|V_P|$ as power of two, hence this check can be done using a simple bitwise right-shift instruction. Depending on the category of the vertex, another right shift instruction is used to compute the actual bucket index. No change is needed in the accumulation thread implementation.

In summary, this optimization distinguishes primary and secondary vertices based on their degrees and the given r_v parameter. Defining two different bucket count parameters (k_p and k_s) for primary and secondary vertices helps achieve a better performance tradeoff between binning and accumulation threads. The actual values for parameters r_v, k_p, and k_s should be determined based on the graph size, vertex degree distribution, and the performance characteristics of the multi-core platform. In the next section, we propose an analytical model to determine these parameters automatically.

6 Analytical Performance Model

As explained earlier, the parallel throughput of the binning threads (denoted as *btpt*) depends on the number of buckets that are being updated at a given time (denoted as k). Similarly, the parallel throughput of the accumulation threads (denoted as *atpt*) depends on the number of vertices per bucket, i.e., the bucket size (denoted as s). For a given application and a target platform, one can model $btpt(k)$ and $atpt(s)$ functions by running the PB algorithm on representative graphs for different k and s values. This characterization can be done only once per multi-core CPU platform.

Given these basic functions, we propose an analytical model to quickly estimate the overall performance of the proposed algorithms for an input graph and a set of input parameters. For this analysis, we will assume the ideal case, where the idle times and synchronization overheads are negligible compared to the total runtime. For simplicity of analysis, we will also assume that binning and accumulation operations are performed by the same set of physical threads, as proposed in Section 3.2.

In this section, we provide our analytical models directly, whereas the detailed derivations and reasonings are given in *Section 3* of the *Supplemental Material*, available online.

The parallel throughput (*tpt*) is defined as the number of edges processed per second, and can be computed for single-level bucketing as follows:

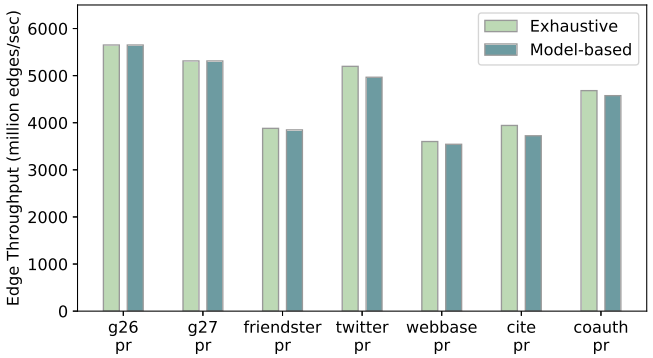

Fig. 5. Throughput comparison of exhaustive and model-based parameter settings for *PageRank* using two-level bucketing.

$$tpt = \frac{1}{\frac{1}{btpt(k)} + \frac{1}{atpt(s)}}. \quad (2)$$

For a given input graph, we can choose the k value that leads to the best estimated performance using Equation (2).

The input parameters that need to be determined for two-level bucketing are the *primary vertex ratio* (r_v) and the number of primary and secondary buckets (k_{pri} and k_{sec}). Let us define an edge as *primary* iff its destination is a primary vertex. For a given graph G, these input parameters also determine the ratio of primary edges (r_e) and the primary and secondary bucket sizes (s_{pri} and s_{sec}).

Note that the working set of a binning thread is likely to include cache lines corresponding to both primary and secondary buckets. As discussed in Section 5, it is expected that the majority of the edges are connected to the primary vertices for scale-free graphs. This implies that the primary buckets are updated more often than the secondary buckets, hence the cache lines corresponding to primary buckets are more likely to remain in local caches. We estimate the binning times involving the primary and secondary buckets by modeling the *effective number of primary* (f_{pri}) and *secondary* (f_{sec}) *buckets for binning operations* as

$$f_{pri} = k_{pri} + (1 - r_e)k_{sec} \quad \text{and} \quad f_{sec} = k_{pri} + k_{sec}. \quad (3)$$

Based on these definitions, we can model the parallel throughput for two-level bucketing as follows (see *Section 3* of the *Supplemental Material*, available online, for detailed derivation):

$$tpt = \frac{1}{\frac{r_e}{atpt(s_{pri})} + \frac{1 - r_e}{atpt(s_{sec})} + \frac{r_e}{btpt(f_{pri})} + \frac{1 - r_e}{btpt(f_{sec})}}. \quad (4)$$

Using this formula, we can quickly estimate the performance of the proposed two-level bucketing algorithm for an input graph and a given set of parameters r_v, k_{pri}, and k_{sec}.

The objective of Equation (4) is not to model the performance in absolute terms. Instead, it is more important to capture the relative change in performance for different sets of parameters so that we can choose the best parameters automatically. To evaluate the effectiveness of the proposed analytical model, we have implemented two parallel graph applications *PageRank (PR)* and *Connected Components (CC)* using our proposed concurrent PB algorithms. Then we have compared two methods of choosing the bucketing parameters:

- *Exhaustive:* We run our algorithm for every single parameter permutation within a range and choose the one with the best performance.
- *Model-based:* We use our analytical performance model defined by Equation (4) to estimate the throughput of computation for each parameter permutation and choose the best one.

Obviously, the exhaustive method gives the set of parameters that leads to the best performance, but it is not practical to run for each input graph. On the other hand, the model-based method can evaluate each set of parameters in constant time by simply plugging in the parameter values in Equation (4).

The throughput results of these two methods for *Page Rank* are shown in Fig. 5. This experiment shows that our model-based automatic parameter tuning methodology leads to performance results within 5 percent of the best results obtained through exhaustive search. The details of this experiment as well the results for the *Connected Components* algorithm are provided in *Section 4* of the *Supplemental Material*, available online.

7 Work Efficiency

7.1 Asynchronous Execution

For graph frameworks that follow the bulk synchronous parallel (BSP) model [17], computations are separated by global barriers [18]. The data updated in the current iteration becomes available only in the next iteration. In contrast, asynchronous execution models allow access to the data updated within the same iteration. This reduces data staleness, allows faster convergence, and leads to significant work efficiency improvements [19], [20], [21].

The original two-phase PB algorithms follow the BSP model, because the binning and accumulation operations of the same iteration are separated by barriers. Specifically, the data of vertices are updated during the accumulation phase, and the updated values are *pushed* to the neighbors in the binning phase of the next iteration.

In contrast, our concurrent execution model has the potential to reduce the staleness of data by adopting the asynchronous model. Since the binning and accumulation threads run concurrently, the vertex data updated during accumulation can be pushed to the neighbors during the binning operations executed afterwards within the same iteration. However, this requires the new vertex data to be up-to-date even after partial accumulation. For example, in Algorithm 3, the *newRank* value of vertex v computed during accumulation will not be ready to use until *all* c_u values from its in-neighbors are processed.

We propose to resolve this issue by adopting the *delta caching* idea proposed in PowerGraph [6], which is based on sending only partial updates to out-neighbors. This idea is illustrated for the PageRank application in Algorithms 4 and 5. Note that the buffer and queue management operations remain the same as in Algorithms 2 and 3, hence they are not repeated here for brevity.

Observe in Algorithm 4 that the Δ_u value for vertex u is computed based on the change in the vertex data since the

last binning operation for u (line 3). Then, this change is propagated to the out-neighbors of u (line 7). During accumulation (Algorithm 5), the partial updates from neighbors are applied to the new data of vertex v (line 3). Next time vertex v is processed by a binning thread, it will propagate these partial updates to its neighbors.

Algorithm 4. Delta Caching Based Binning for PageRank

1: **for** each vertex u in V_T **do**
2: $newRank \leftarrow newRanks[u]$ *// store in a local variable*
3: $\Delta_u \leftarrow \beta \times (newRank - oldRanks[u])/d_u$
4: $oldRanks[u] \leftarrow newRank$
5: **for** each out-neighbor v_i of u **do**
6: $k \leftarrow Bin(v_i)$ *// destination bucket*
7: add (Δ_u, v_i) to intermediate buffer k
// buffer operations are the same as in Algorithm 2
8: **end for**
9: **end for**

Algorithm 5. Delta Caching Based Accumulation for PageRank

1: $C \leftarrow$ *chunk dequeued for accumulation as in Algorithm 3*
2: **for** each (Δ_u, v) in C **do**
3: $newRanks[v]$ += Δ_u
4: **end for**

Note that no race condition is introduced to implement delta caching in our framework. It is guaranteed that only one thread can update the old or new data of a vertex at a given time. However, the new vertex data read by a binning thread might be updated by an accumulation thread at the same time. To guarantee correct execution of the binning thread, we first load the new vertex data to a local variable[6] (line 2 of Algorithm 4). This ensures that the new data value used in different lines of the binning thread is consistent.

We will show in our experiments that enabling asynchronous execution in the PB algorithm can improve work efficiency significantly.

7.2 Active Vertex Set Support

In *data-driven* graph algorithms, the execution begins with a set of *active* vertices, which is updated during execution based on the computations performed [22]. For example, the set of active vertices for PageRank includes all vertices initially. As the rank values of the vertices are converged over multiple iterations, they can be removed from the active set so that only the unconverged vertices are processed. Another example is the single source shortest path (SSSP) algorithm, where only the source vertex is in the active set initially. As the frontier is expanded, other vertices are added to or removed from the active set. For work efficiency, it is important to process only the vertices in the active set in each iteration, rather than all vertices.

It was shown that many graph applications can be modeled as a matrix-vector multiplication operation [1]. Using a dense vector as in SpMV corresponds to processing all vertices, whereas a sparse vector as in SpMSpV corresponds to processing only the set of active vertices. The PB-based SpMV algorithm was extended to SpMSpV in a recent work [5]. For this, the authors have used a data structure called sparse accumulator (SPA) [23], which is an abstract type that consists of a dense data vector and a list of indices that refer to the nonzero entries in the sparse vector. During the accumulation phase, the SPA is updated and the list of unique nonzero indices (which will become the *active set* of the next iteration) is maintained. Since each bucket is processed by a different thread, SPA updates are done independently by multiple threads.

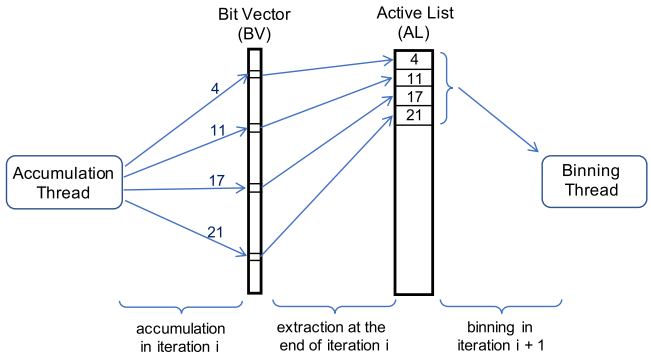

Fig. 6. Active set operations for one bucket are illustrated on a simple example, where vertices 4, 11, 17, and 21 are activated in iteration i by an accumulation thread and processed in iteration $i+1$ by a binning thread.

Active set support can be added to our proposed concurrent PB algorithm in a similar way. In our implementation, we have defined an active set data structure per bucket. Our active set consists of a bit vector (denoted as BV), where each bit corresponds to a vertex in the bucket, and a list of indices corresponding to the active vertices (denoted as AL). Then, the binning and accumulation operations given in Algorithms 4 and 5 are modified as follows. Line 1 of Algorithm 4 is changed such that the binning operations are done only for the vertices in the ALs. Then, in the accumulation threads, when the data of a vertex v is updated (line 3 of Algorithm 5), the index of v is set to 1 in the corresponding BV. Finally, at the end of each iteration, the following operations are performed for each (BV, AL) pair: 1) AL is cleared, 2) the set bits of the BV are extracted and the corresponding nonzero indices are added to the AL, and 3) the BV is cleared. Since there is no data dependency between (BV, AL) pairs of different buckets, these operations are done in parallel by different threads.

Fig. 6 illustrates these operations for one bucket on a simple example. In iteration i, four vertices are activated during accumulation operations by setting the corresponding bit vector entries to 1. At the end of iteration i, the set bits are extracted from BV and the corresponding indices are stored in sorted order in AL. In the next iteration, the binning thread in charge of this bucket processes only the list of vertices in AL. In other words, while a binning thread reads from AL that was populated in the previous iteration, an accumulation thread writes to BV of the current iteration. Furthermore, there is a separate (BV, AL) pair for each bucket, and it is guaranteed that there can be at most one accumulation thread active for a particular bucket (Section 3). Hence, the active set operations are performed without race conditions.

6. This can be achieved by using the *volatile* keyword in C++.

TABLE 1
Graphs Used in the Experiments

Graph	Description	# vtxs (M)	# edges (M)	avg degree
g26	Kronecker synthetic [8]	67	2,104	31
g27	Kronecker synthetic [8]	134	4,223	31
friendster	Social network [24]	125	1,806	14
webbase	Web graph [25]	118	993	8
twitter	Social network [26]	62	1,468	23
coauth	Coauthorship network [27]	115	1,696	14
cite	Citation network [27]	47	529	11

8 Experimental Setup

We have performed our experiments on a 2-socket Xeon CPU E5-2650-v4 system with 24 cores (hyperthreading disabled) and 128 GB DRAM. We have implemented the proposed algorithms using C++ and compiled with g++ version 5.4.0 using optimization flag -O3.

For our experiments, we have chosen the scale-free graphs used by the original PB algorithm [3]. These graphs are listed in Table 1. Note that two of these graphs (*g26* and *g27*) are Kronecker graphs that are similar to the input graphs of Graph500 and are synthetically generated using the software provided in the Berkeley GAP Benchmark Suite [8]. The others in Table 1 are real-world graphs provided by the corresponding references. For consistency, we have randomly shuffled the vertex indices of these graphs to remove any potential effects due to earlier preprocessing (such as locality optimization) that might have been done by the corresponding references.

8.1 Graph Applications

We differentiate two types of execution modes. In the high throughput (HT) execution mode, the input graph is assumed to be static and all vertices are processed in every iteration in a deterministic way. On the other hand, only the necessary subset of vertices and edges are processed in different iterations of the work efficient (WE) execution mode. We have implemented and experimented with two applications for each mode as summarized below.

PageRank - High Throughput. This algorithm is the running example used throughout the paper, so no further details are provided here.

Connected Components - High Throughput. The objective of this application is to identify and label the weakly connected components in a directed graph. Since our paper focuses on the vertex-centric graph applications, we have chosen the parallel Hash-Min PPA algorithm [28]: Initially, each vertex is defined to have a label equal to its index. Then, for each vertex v, the minimum label value is computed among the vertices u that have an edge to v. If the label of v is greater than this minimum label, then it is updated to this minimum value. The iterations continue until all vertices in a connected component end up with the same label value.

PageRank - Work Efficient. L_∞ metric is used to minimize the maximum error among all vertices [29]. In our implementation, we have chosen a relative error threshold of 0.1 percent to determine convergence. If the page rank value of vertex u changes by less than the relative error threshold, then it is defined to be converged. Otherwise, the change in u's rank value is *propagated* to u's out-neighbors as explained in Section 7.1. If a vertex v receives an updated value from a neighbor, it is added to the active set of the next iteration. Iterations continue until the active set is empty.

Single Source Shortest Path - Work Efficient. Given a weighted directed graph and a source vertex s, the objective is to compute the distance of each vertex to s. We use the parallel and work-efficient variant of Bellman-Ford algorithm, because it can be represented as an SpMSpV operation. Initially, the active set contains only the source vertex s. In each iteration, the well-known edge relaxation operation [30] is performed for each outgoing edge of each vertex u in the active set. If the distance value of a vertex v is updated, it is added to the active set of the next iteration. We have used the active set data structure described in Section 7.2 in our implementation.[7]

For all these applications, we have used the same basic code infrastructure to implement the operations proposed in this paper. Only the kernel operations per vertex and edge in the binning and accumulation threads are implemented specific to each application.

8.2 Baseline Implementations

Since the original two-phase PB algorithm [3] is not publicly available, we have implemented it starting with the codebase provided in the GAP Benchmark Suite [8]. In addition to the low-level optimizations proposed in [3], we have implemented non-uniform memory access (NUMA) optimizations to allow more efficient DRAM accesses to the bucket data.[8] The baseline two-phase PB implementation, which includes all these optimizations, will be referred to as *2-phase* in the experimental results.

To validate the performance of our baseline implementation, we have compared the throughput improvements reported for the original PB algorithm with our experimental results. In [3], the pull-based PageRank implementation from the GAP Benchmark Suite was chosen as the baseline after showing that it outperformed implementations from Galois [32], GraphMat [33], Ligra [34], and Compressed Sparse Blocks [35]. It was shown that the performance of the PB algorithm was better than the pull-based PageRank algorithm by a factor of 1.5-2x on most of the benchmark graphs (*webbase* was the only exception with a performance improvement of $\sim$2.5x over the pull-based algorithm). These results were reported as a bar chart in Fig. 4 of [3].

We have downloaded the same pull-based PageRank implementation [8] and compared its performance with our baseline PB implementation on our experimental platform. The results are shown in the top half of Fig. 7, where the numbers on the bars are the speed up values over the pull-based PageRank implementation. Observe that our baseline implementation (*2-phase*) has performance improvements in the range 1.7-2.5x on the same set of graphs. In general, the

7. It is also possible to use a priority-ordered active set implementation (similar to the one used for the parallel delta stepping algorithm [31]) to improve work efficiency further. However, this is an orthogonal issue which is not within the scope of this paper.

8. We have observed performance improvements when the accumulation threads access the bucket data stored in their local DRAMs.

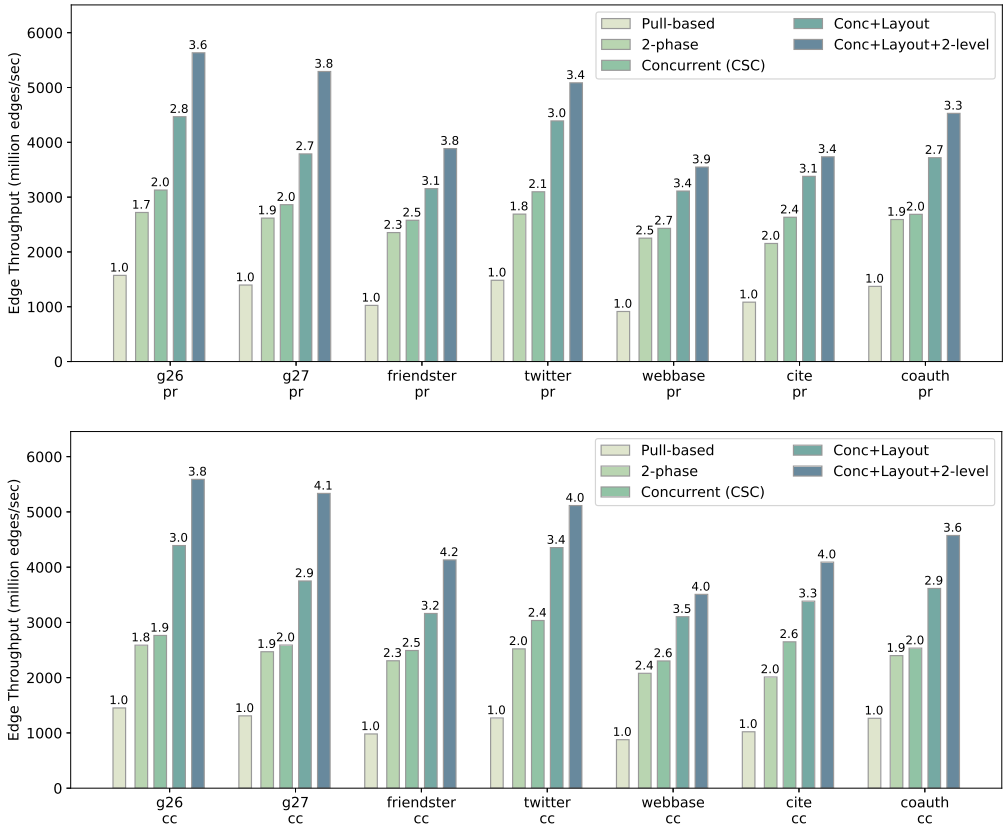

Fig. 7. Throughput of computation for different *PageRank* (pr) and *Connected Components* (cc) implementations. The speed up values over the pull-based algorithm are marked on the bars. (Higher values indicate higher throughput.).

speed up values obtained by our baseline implementation are similar to (if not slightly better than) the speed up values reported in Fig. 4 of [3]. Hence, we believe that our *2-phase* implementation is a reasonable baseline that reflects the performance of the original PB algorithm.

The work efficient version of the PB algorithm was proposed as an SpMSpV operation in a separate work [5], the code of which is available publicly as part of the Combinatorial BLAS repository [36]. As proof of concept, the authors of [5] have provided an implementation of the breadth-first search (BFS) algorithm using their proposed SpMSpV framework. We have downloaded and experimented with that BFS code and compared it with our *2-phase* baseline implementation that includes active set support. Unfortunately, the SpMSpV-based implementation failed to run on three of the large graphs (*friendster*, *twitter*, and *g27*). So, we generated three smaller Kronecker graphs as shown in Table 2. Observe that our baseline implementation is significantly faster than the code provided for [5], probably due to the fact that the SpMSpV implementation was not well-tuned for large and scale-free matrices. For this reason, we will use our 2-phase implementation with active set and delta caching support as the baseline for the work efficient experiments.

These results show that our baseline implementation (*2-phase*) has comparable or better performance compared to state-of-the-art PB implementations in both high-throughput and work-efficient execution modes.

TABLE 2
Comparison of Baseline Implementations for Work-Efficient Execution

Graph	# vertices (M)	# edges (M)	SpMSpV [5] runtime (s)	2-phase PB runtime (s)
g23	8	259	2.77	0.14
g24	17	521	5.08	0.29
g25	34	1,047	10.26	0.62
g26	67	2,104	20.89	1.36
webbase	118	993	34.60	1.85
cite	47	529	16.99	0.38
coauth	115	1,696	20.15	1.42

8.3 Execution Parameters

For the baseline *2-phase* implementation, the main parameter that has an impact on performance is k, the number of buckets. In our experiments, we have run the baseline implementation for all k values in the set $\{2^i \mid i \in [7, 12]\}$. Then, for each graph, we have chosen the k value that resulted in the best baseline performance.

In contrast, for our proposed algorithms, we have chosen the bucketing parameters using the analytical performance model proposed in Section 6. In other words, while the results reported for the baseline algorithm are the best ones obtained after parameter enumeration, the results reported for the proposed algorithm are based on the bucketing parameters that are chosen automatically. Note that the proposed algorithm also has an additional parameter, *size of one chunk*, which controls the total size of the intermediate buffers[9] as described in

9. As mentioned in Section 3.2, we have fixed the number of buckets to be equal to 64 so that the intermediate buffer size is controlled by only the *chunk-size* parameter.

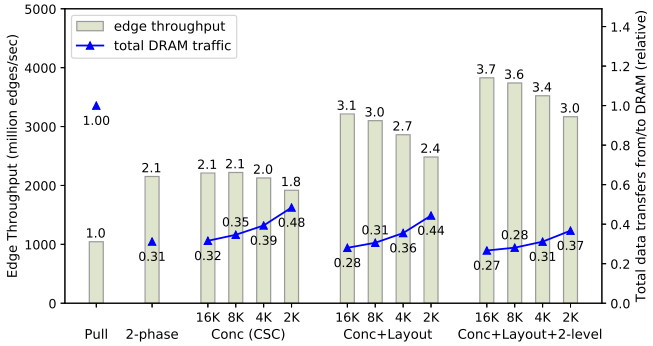

Fig. 8. Throughput (bars and the left y-axis) and total DRAM traffic (lines and the right y-axis) of different *PageRank* implementations for the *friendster* dataset. The values on the bars show the speedup over the pull-based algorithm. The values next to the lines show the total DRAM bandwidth utilization normalized with respective to the pull-based implementation. The concurrent implementation results are shown for different chunk-size parameters shown in the x-axis.

Section 3.2. We will show in our experiments that a tradeoff can be obtained between performance and memory size by changing this parameter.

In the high-throughput execution experiments, each run consists of 100 iterations, and the performance metric is the throughput of computation, i.e., the number of edges processed per second. On the other hand, runtime is the main performance metric for the work-efficient execution experiments, where the iterations continue until convergence. Each experiment is repeated 10 times and the reported results are the average of 10 measurements.

9 Experimental Results

9.1 High Throughput Execution

In the first experiment, we have evaluated the impact of the different features that we have proposed in this paper. The results are reported in Fig. 7 for the Pagerank and Connected Components applications.

Here, the pull-based implementation of PR is from the GAP Benchmark Suite [8]. We have also modified this code in a straightforward way to implement the pull-based CC algorithm. The *2-phase* results are from the baseline implementation discussed in Section 8.2, and they represent the performance of the state-of-the-art PB algorithm. Note that the speed-up values reported for the 2-phase algorithm over the pull-based implementation [8] are consistent with the improvements reported in the original PB work [3].

Concurrent (CSC) is the algorithm we propose in Section 3 and it is directly applicable to the traditional CSC graph format. Note that this version includes the packing optimization for the intermediate buffer storage (described in Section 4) since this optimization is independent of the input edgelist format. For this experiment, we have set the size of each chunk to be 16K entries. Observe that despite using much less intermediate storage for the bucket data, its performance is similar to the *2-phase* baseline algorithm in general.

Conc+Layout is the concurrent PB algorithm that uses the optimized graph layout we propose in Section 4, which includes optimizations for both input edgelist and intermediate buffer storage. In addition to reducing the memory storage requirements, this graph layout improves performance significantly due to the 4-bit local buffer pointers stored in the edgelist. These 4-bit pointers allow efficient binning operations as discussed in Section 4.

Finally, *Conc+Layout+2-Level* is the implementation that includes the two-level bucketing idea we propose in Section 5 in addition to the optimized graph layout. It can be observed that two-level bucketing is another feature that improves the throughput of computation significantly.

The results of this experiment show that while the original 2-phase PB algorithm improves the throughput of computation by about twice compared to the pull-based implementations, our proposed ideas lead to throughput improvements of up to four times.

The purpose of the next experiment is to provide details on how different implementations affect the amount of data transferred from/to DRAM[10] (Fig. 8). In this experiment, we vary the chunk size parameter (from $16K$ downto $2K$ entries per chunk) for the concurrent implementations based on the discussions in Section 3.2.

Observe that the 2-phase PB algorithm reduces the total DRAM traffic of the original pull-based algorithm by almost 70 percent. This is due to the access locality improvements of the PB algorithm. As stated before, the proposed concurrent execution model reduces the total memory storage size, but it has the potential to increase the amount of cold cache misses, hence the total DRAM traffic. The results in Fig. 8 show that there is only minimal such increase if the chunk size parameter is chosen large enough (e.g., $8K$ or $16K$ entries per chunk). This confirms the intuition about expected amortization of cache misses as explained in last paragraph of Section 3.3. However, as the chunk size parameter is reduced further, the amortization factor is expected to decrease, which leads to increased DRAM traffic and less speedup.

The layout optimization feature reduces the intermediate buffer storage size significantly, but its effect on total DRAM traffic is limited as can be seen by comparing the DRAM traffic of *Conc (CSC)* and *Conc+Layout* implementations. The main reason is that the total DRAM traffic is mostly dominated by the writes/reads of vertex data to/from buckets, as well as cold misses described above for small chunk sizes, which are unaffected by layout optimizations. However, it is still interesting to note that with layout optimization and chunk size parameter of 16K, the proposed algorithms use slightly less DRAM traffic compared to the 2-phase algorithm.

In the next experiment, we evaluate the effectiveness of our proposed algorithms in terms of both performance and memory overhead by showing the impact of the chunk size parameter discussed in Section 3.2. Fig. 9 shows the results of this experiment for PR and CC applications.

Here, the values on the bars show the relative speed up over the baseline 2-phase algorithm, whereas the values next to the lines show the relative memory overhead with respect to the original graph size. As an example, for application PR and graph *g27*, the baseline 2-phase algorithm requires an *additional* memory storage of 177 percent for its

10. The total DRAM traffic was measured using Intel's VTune software and uncore performance counters related to the integrated memory controller.

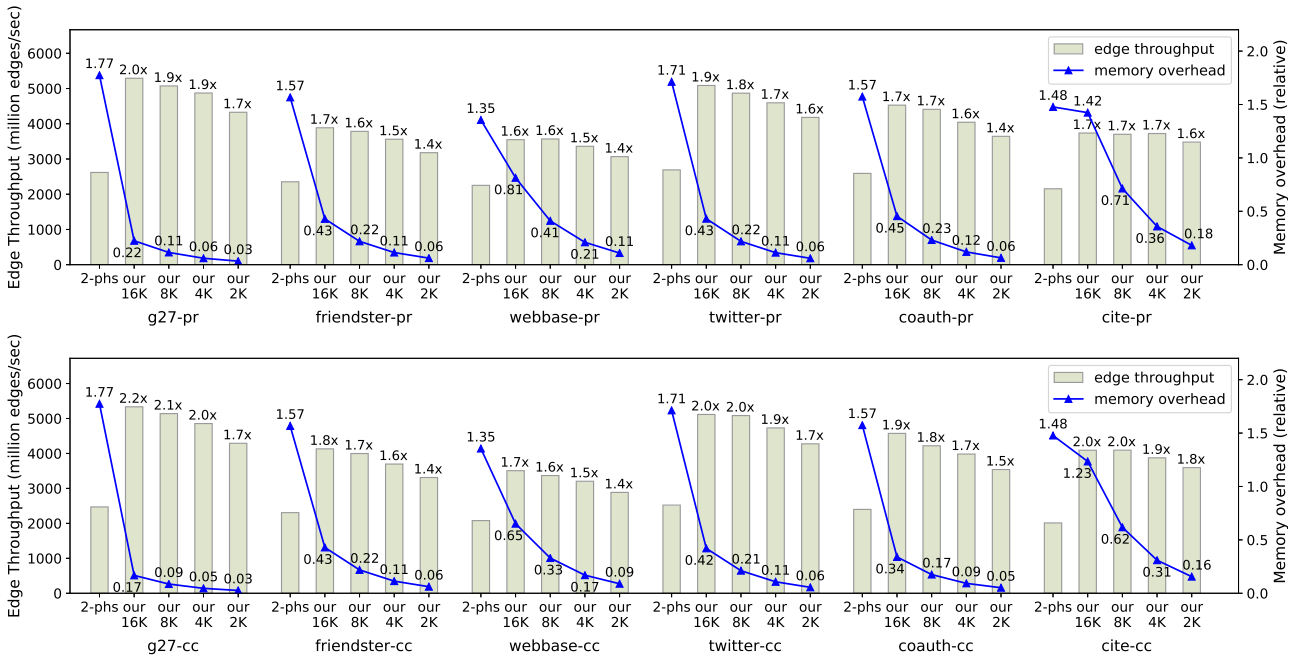

Fig. 9. Tradeoff between throughput of computation (bars and the left y-axis) and memory storage overhead (the lines and the right y-axis) controlled by the chunk size parameter for high-throughput *PageRank* (pr) and *Connected Components* (cc). Our implementation includes the proposed layout and 2-level bucketing features. The values on the bars show the speedup over the 2-phase algorithm. The values next to the lines show the relative memory overhead with respect to the original graph size.

intermediate bucket data. On the other hand, our proposed concurrent algorithm requires only 22 percent extra memory storage (with respect to the original CSC graph) when the chunk size parameter is set to 16K entries, and it leads to 2x better throughput compared to the 2-phase algorithm. As the chunk size parameter is decreased to 8, 4, and 2K, the relative memory overhead reduces to 11, 6, and 3 percent, while the throughput improvement stays relatively high at 1.9x, 1.9x, and 1.7x, respectively.

This experiment shows that our concurrent PB algorithm can reduce the memory overhead substantially (e.g., from 177 to 3 percent of the original graph size), while improving the performance of the baseline algorithm by up to 2.2x.

9.2 Work Efficient Execution

In this section, we perform experiments to evaluate the effectiveness of our concurrent PB algorithm in the context of work efficient execution. Since an active set is maintained to determine the vertices that need to be processed, the execution is not deterministic across iterations. Hence, the destination vertex indices need to be written to buckets in every iteration, and the optimized graph layout proposed in Section 4 is not applicable for this execution mode. In other words, the same CSC input graph format is used for both the baseline and the proposed PB implementations. Both baseline 2-phase and our concurrent implementations include delta-caching and active set support, and our implementation includes the proposed 2-level bucketing feature.

As explained in Section 7.1, the 2-phase PB algorithm follows the bulk synchronous model because the binning and accumulation operations are separated by barriers, whereas the proposed PB algorithm can implement the asynchronous model due to concurrent execution. In the next experiment, we evaluate the work efficiency benefits of asynchronous execution for PR and SSSP applications in Fig. 10.

In this figure, the values on the bars show the throughput improvements over the baseline 2-phase algorithm, and the values marked next to the lines show the amount of work done (i.e., the total number of edges processed) relative to the baseline. As an example, for the *SSSP* application and the *friendster* graph, when the chunk size is $16K$, $1.8x$ better throughput is achieved using our algorithm and the amount of work done is 77 percent of the baseline, i.e., 23 percent less work is done due to asynchronous execution. As the chunk size is reduced down to $2K$, the amount of work done decreases down to 58 percent, while the throughput improvement reduces to $1.5x$.

These results show that concurrent execution not only reduces the memory overhead, but also improves work efficiency due to asynchronous execution. As the intermediate buffer size is reduced, there is less staleness of data between binning and accumulation steps, hence the convergence behavior is improved. The work efficiency improvements vary among the applications and the graphs. However, they are consistent in general with the previously reported improvements due to asynchronous execution [10], [29].

In terms of overall runtime, there is a tradeoff between throughput and work efficiency. As the chunk sizes are decreased, the throughput tends to degrade while the work efficiency improves. In the next experiment, we study the effect of intermediate buffer size on the overall runtime. These results are shown in Fig. 11.

Here, the memory overhead is given relative to the original CSC graph size plus the size of the active set, which is needed for work-efficient execution. Notice that the relative memory overheads for the *SSSP* application are in general smaller than *PageRank*, because SSSP uses a weighted graph, hence the size of the original CSC graph is larger. The speed-up values shown here are with respect to the runtime of the baseline 2-phase algorithm, which uses the same

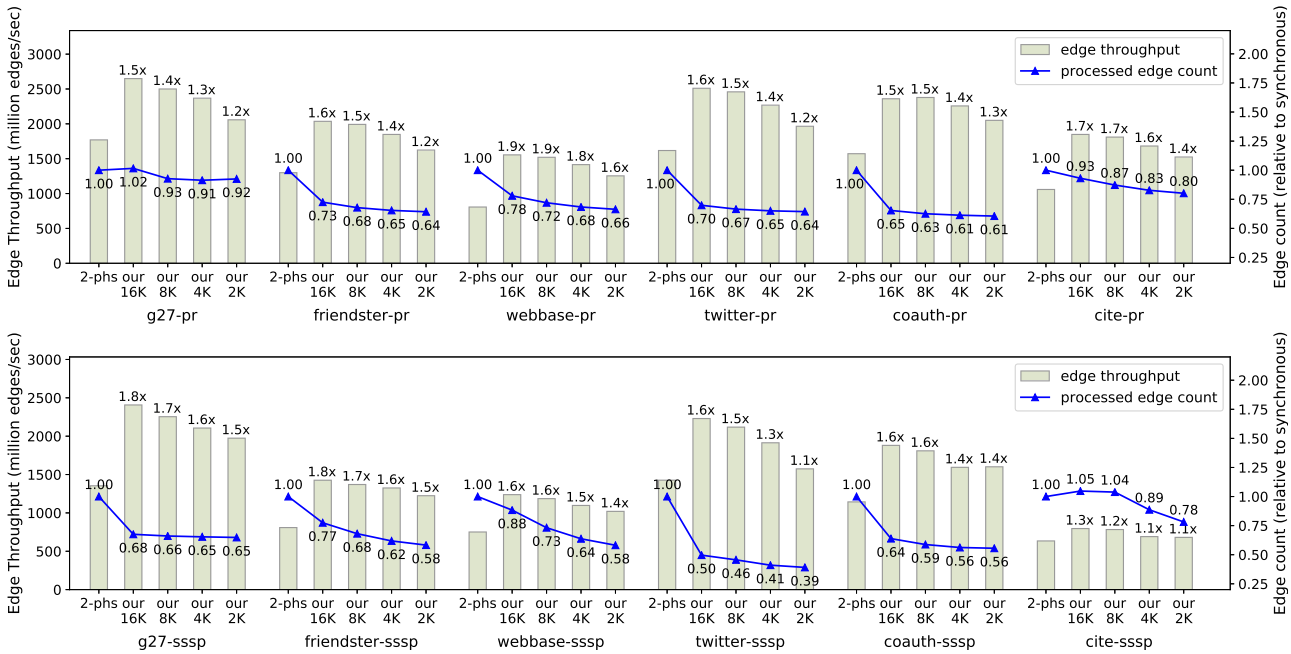

Fig. 10. Evaluation of throughput (bars and the left y-axis) and the total number of edges processed (lines and the right y-axis) for work-efficient *Page Rank* (pr) and *single source shortest path* (sssp).

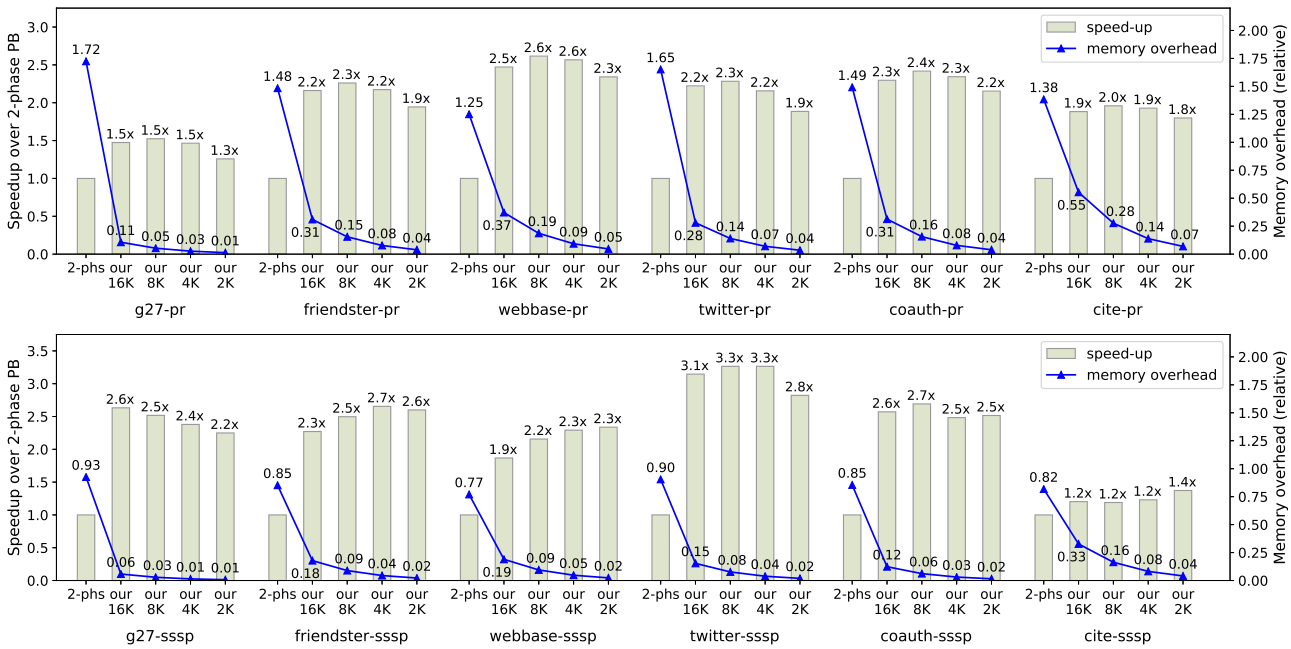

Fig. 11. Tradeoff between speed-up (bars and the left y-axis) and memory overhead (the lines and the right y-axis) controlled by the chunk size parameter for *work-efficient PageRank* (pr) and *single-source shortest path* (sssp).

active set data structure (larger values indicate better speed-up).

While the baseline 2-phase algorithm stores a *(data, index)* pair for each edge in the graph, our proposed algorithm maintains small buffers and processes them concurrently as they are generated by the binning threads. This approach not only reduces the memory storage overhead, but also improves work efficiency. The results in Fig. 11 show that the memory overhead of our concurrent algorithm can be reduced to less than 5 percent of the original graph, while still obtaining significant speed up over the baseline 2-phase algorithm.

10 Related Work

Vertex centric graph algorithms are defined based on a computational kernel that is executed on each vertex. Typically, execution of a vertex program on vertex v requires access to the local data of v as well as the data of v's neighbors. Several shared-memory parallel and distributed graph frameworks have been proposed based on this model, such as Pregel [18], Ligra [34], Graphlab [6], [9], and Galois [32]. For more details on vertex centric graph frameworks, readers can refer to a recent survey paper [37].

Yan, et al. identified a set of desirable properties for a vertex-centric algorithm to have good performance [28]. Specifically, an algorithm is defined to be *balanced practical Pregel algorithm (BPPA)* if it satisfies the following constraints: 1) each vertex uses space proportional to its degree, 2) the computation cost for each vertex is proportional to its degree, 3) the size of messages sent/received by each vertex is proportional to its degree, 4) the number of supersteps is at most logarithmic in the number of vertices. We expect that the techniques we propose in this paper will potentially be useful for graph algorithms that satisfy constraints 1-3.

Vertex-centric graph algorithms are also commonly represented as linear algebra operations [1], especially in terms of generalized sparse matrix dense/sparse vector multiplication (SpMV/SpMSpV) kernels [33], [38]. Since the techniques we have proposed are also applicable to SpMV/SpMSpV kernels, the graph algorithms modeled using these kernels have the potential to benefit from our proposed improvements.

Parallel performance of graph algorithms depend on both throughput of computation and work efficiency [10]. Existing works that use SpMV-based models typically aim to improve throughput of computation. However, in iterative algorithms, vertices may converge at different speeds, which is known as asymmetric convergence [9]. Similarly, traversal-type algorithms are based on processing a frontier of vertices in every iteration. In general, these types of algorithms are called *data driven* [39], and they can also be modeled as SpMSpV operations to enable work efficient execution.

Access to the neighbor vertex data is typically the main throughput bottleneck for shared-memory parallel graph implementations. For large graphs, all vertex data cannot fit into the last level cache (LLC) of the system. In the absence of locality, accesses to neighbors' data lead to expensive DRAM accesses. Cache blocking is a technique to improve cache locality by partitioning the graph into smaller blocks [40]. However, the experiments by Beamer, et al. have demonstrated that cache blocking is not effective for large scale-free graphs [3]. Others have proposed vertex reordering techniques based on hypergraph partitioning to improve temporal and spatial locality [41], [42]. These techniques require expensive preprocessing operations. Furthermore, it is known that scale-free graphs are hard to partition and the partitioning benefits are limited.

To improve work efficiency, asynchronous parallel execution model was proposed for graph algorithms, where the vertices can access the latest available data of their neighbors. It was shown that reducing the staleness of data can lead to faster convergence for several graph applications [9], [10], [19], [20], [21]. Data driven implementations were also proposed to improve work efficiency. The basic idea is to maintain an active set to determine the vertices that will be processed in every iteration [33], [43], [44].

Delta-stepping is another algorithm that uses bucket data structures for single-source shortest path computations [8], [31], [45]. The objective of bucketing in delta stepping is to determine the vertex priorities based on radix-sorted distance values, whereas we use bucketing in this paper to improve memory access efficiency. However, it is possible to extend our active set implementation (Section 7.2) to support priority-order by using ideas similar to delta-stepping.

Propagation blocking algorithm was proposed recently for PageRank [3] and SpMV [4]. This algorithm was also extended to SpMSpV to enable work efficient execution [5]. These algorithms have been explained in detail in Sections 2 and 7 of this paper. They were shown to be effective especially for shared-memory parallel execution and large and scale-free graphs. Our paper improves upon them in several aspects, including the throughput of computation, memory storage, and work efficiency.

11 Discussions and Conclusions

Propagation blocking is a recently proposed parallel algorithm for shared memory systems for the purpose of alleviating DRAM access bottlenecks of graph algorithms for large and irregular data. In this paper, we have proposed several improvements in terms of throughput, work efficiency, and memory storage overhead. Our experiments show that while the original PB algorithm leads to up to 2.5x parallel speedup over the pull-based algorithms, our proposed improvements provide up to 4.2x speedup on a 24-core Xeon system.

Furthermore, since the binning and accumulation threads run concurrently in our algorithm, the intermediate data storage can be reduced significantly. Our results have shown that while the original PB algorithm requires additional memory of size up to 177 percent of the original graph, the memory overhead of our algorithm can be reduced down to less than 5 percent of the original graph size, while still having significant performance improvements over the baseline. This is especially important to be able to process very large graphs on a shared-memory multi-core system with limited DRAM size.

Our experiments also show that the concurrent execution model not only helps reduce the memory storage overhead, but also improves work efficiency because it enables asynchronous parallel (AP) execution. It is well known that AP execution reduces staleness of data and leads to faster convergence compared to the bulk synchronous parallel execution. While the original PB algorithm was inherently based on the BSP model (due to well-separated phases of binning and accumulation), we have shown that asynchronous execution can be enabled for PB without sacrificing parallel performance.

Our proposed algorithms are applicable to both high-throughput and work-efficient execution modes. For HT execution, the same set of vertices are processed every iteration, so the vertex indices do not need to be written to buckets in every iteration, as also was pointed by the previous works [3], [4]. In this paper, we have proposed an optimized graph layout, where only 4-bit local buffer pointers are stored in the CSC edgelist to improve performance of the binning threads. Furthermore, we have proposed a simple packing scheme that leads to storage overhead of only a few bits per edge. Our experiments have shown that these techniques not only reduce the memory overhead, but also improve the throughput of computation significantly.

For WE execution mode, we have incorporated the *delta caching* idea [6] and the *active set* data structure into the PB algorithm. Our results show that these techniques help improve work efficiency. Combined with our proposed

throughput improvements, we obtain up to 3.3x parallel speedup over the baseline PB algorithm that has similar delta caching and active set features implemented.

The proposed concurrent PB algorithm can be run directly on an input CSC graph without any preprocessing. However, we have shown that performance can be improved further if our optimized graph layout is used. This layout can be generated on-the-fly as the PB algorithm is executed on the input graph in the first iteration. The other proposed improvement, two level bucketing, requires partitioning the vertices into two groups: high-degree vertices and low-degree vertices, which can be done easily during graph construction. Note that our proposed work-efficient algorithm requires no preprocessing and can be executed on a CSC graph directly.

In this paper, we have focused on vertex-centric graph applications that can be modeled as SpMV or SpMSpV operations. The techniques we have proposed are applicable to a range of graph applications. In our experiments, we have evaluated *PageRank*, *connected components*, and *single-source shortest path* algorithms. We expect to see similar benefits for *breadth-first traversal* and the algorithms that use it as building block such as *betweenness centrality* and *bipartite matching*. Other examples that follow a similar computational pattern include *loopy belief propagation* [9], [11], maximal independent set [28], and graph coloring [46].

We have used the same code infrastructure to implement different graph applications by simply changing the computation kernels. As future work, it is possible to create a parallel graph framework that encapsulates the common operations and allows the users to develop different applications by defining the basic computational kernels.

ACKNOWLEDGMENTS

This work was partially supported by TUBITAK 2232 Program under grant number 116C079.

REFERENCES

[1] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. Philadelphia, PA, USA: SIAM, 2011.
[2] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *Proc. Int. Conf. World Wide Web*, 2011, pp. 607–614.
[3] S. Beamer, K. Asanovic, and D. Patterson, "Reducing PageRank communication via propagation blocking," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2017, pp. 820–831.
[4] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T. Tuan, "Optimizing sparse matrix-vector multiplication for large-scale data analytics," in *Proc. Int. Conf. Supercomput.*, 2016, Art. no. 37.
[5] A. Azad and A. Buluc, "A work-efficient parallel sparse matrix-sparse vector multiplication algorithm," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2017, pp. 688–697.
[6] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 17–30.
[7] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Stanford InfoLab, 1999.
[8] S. Beamer, K. Asanovi, and D. Patterson, "The GAP benchmark suite," arXiv:1508.03619, 2015.
[9] 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 Endowment*, vol. 5, pp. 716–727, 2012.
[10] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, S. M. Burns, and O. Ozturk, "Architectural requirements for energy efficient execution of graph analytics applications," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2015, pp. 676–681.
[11] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. M. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 166–177.
[12] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "To push or to pull: On reducing communication and synchronization in graph computations," in *Proc. Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2017, pp. 93–104.
[13] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an Ivy Bridge server," in *Proc. IEEE Int. Symp. Workload Characterization*, 2015, pp. 56–65.
[14] A. Yasin, "A top-down method for performance analysis and counters architecture," in *Proc. IEEE Symp. Perform. Anal. Syst. Softw.*, 2014, pp. 35–44.
[15] X. Wang and G. Chen, "Complex networks: Small-world, scale-free and beyond," *IEEE Circuits Syst. Mag.*, vol. 3, no. 1, pp. 6–20, 2003.
[16] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 2009, ch. 7.
[17] L. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.
[18] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.
[19] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in Pregel-like graph processing systems," *Proc. VLDB Endowment*, vol. 8, pp. 950–961, 2015.
[20] Y. Tian, A. Balmin, S. Corsten, S. Tatikonda, and J. McPherson, "From "think like a vertex" to "think like a graph"," *Proc. VLDB Endowment*, vol. 7, pp. 193–204, 2013.
[21] G. Wang, W. Xie, A. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy," in *Proc. Conf. Innovative Data Syst. Res.*, 2013, pp. 3–6.
[22] R. Nasre, M. Burtscher, and K. Pingali, "Data-driven versus topology-driven irregular computations on GPUs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2013, pp. 463–474.
[23] J. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in MATLAB: Design and implementation," *SIAM J. Matrix Anal. Appl.*, vol. 13, no. 1, pp. 333–356, 1992.
[24] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowl. Inf. Syst.*, vol. 42, no. 1, pp. 181–213, 2015.
[25] T. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, 2011, Art. no. 1.
[26] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter a social network or a news media?" in *Proc. Int. World Wide Web Conf.*, 2010, pp. 591–600.
[27] A. Sinha, Z. Shen, Y. Song, H. Ma, D. Eide, B. Hsu, and K. Wang, "An overview of Microsoft academic service (MAS) and applications," in *Proc. 24th Int. Conf. World Wide Web*, 2015, pp. 243–246.
[28] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu, "Pregel algorithms for graph connectivity problems with performance guarantees," *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 1821–1832, 2014.
[29] A. Arasu, J. Novak, A. Tomkins, and J. Tomlin, "PageRank computation and the structure of the web: Experiments and algorithms," in *Proc. 11th Int. World Wide Web Conf.*, 2002, pp. 107–117.
[30] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 2009, ch. 24.
[31] U. Meyer and P. Sanders, "δ-stepping: A parallelizable shortest path algorithm," *J. Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.
[32] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proc. Symp. Operating Syst. Principles*, 2013, pp. 456–471.
[33] N. Sundaram, N. R. Satish, M. M. A. Patwary, S. R. Dulloor, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High performance graph analtyics made productive," *Proc. VLDB Endowment*, vol. 8, pp. 1214–1225, 2015.
[34] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2013, pp. 135–146.
[35] A. Buluc, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrixtranspose-vector multiplication using compressed sparse blocks," in *Proc. Symp. Parallelism Algorithms Archit.*, 2009, pp. 233–244.

[36] A. Buluc and J. R. Gilbert, "The combinatorial BLAS: Design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, pp. 496–509, 2011.
[37] R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for distributed graph processing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, p. 25, 2015.
[38] J. Kepner, P. Aaltonen, D. Bader, A. Bulu, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, and S. McMillan, "Mathematical foundations of the GraphBLAS," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2016, pp. 1–9.
[39] J. Whang, A. Lenharth, I. Dhillon, and K. Pingali, "Scalable data-driven pagerank: Algorithms, system issues, and lessons learned," in *Proc. Eur. Conf. Parallel Process.*, 2015, pp. 438–450.
[40] R. Nishtala, R. Vuduc, J. Demmel, and K. Yelick, "When cache blocking of sparse matrix vector multiply works and why," *Applicable Algebra Eng. Commun. Comput.*, vol. 18, pp. 297–311, 2007.
[41] K. Akbudak, E. Kayaaslan, and C. Aykanat, "Hypergraph partitioning based models and methods for exploiting cache locality in sparse matrix-vector multiplication," *SIAM J. Sci. Comput.*, vol. 35, no. 3, pp. C237–C262, 2013.
[42] M. Karsavuran, K. Akbudak, and C. Aykanat, "Locality-aware parallel sparse matrix-vector and matrix-transpose-vector multiplication on many-core processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 6, pp. 1713–1726, Jun. 2016.
[43] C. Yang, Y. Wang, and J. Owens, "Fast sparse matrix and sparse vector multiplication algorithm on the GPU," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshop*, 2015, pp. 841–847.
[44] E. Nurvitadhi, A. Mishra, Y. Wang, G. Venkatesh, and D. Marr, "Hardware accelerator for analytics of sparse data," in *Proc. Conf. Des. Autom. Test Eur.*, 2016, pp. 1616–1621.
[45] K. Madduri, D. Bader, J. Berry, and J. R. Crobak, "An experimental study of a parallel shortest path algorithm for solving large-scale graph instances," in *Proc. Meet. Algorithm Eng. Expermiments*, Jan. 2007, pp. 23–35.
[46] A. Khan, "Vertex-centric graph processing: The good, the bad, and the ugly," in *Proc. Int. Conf. Extending Database Technol.*, 2017, pp. 438–441.

Muhammet Mustafa Ozdal received the PhD degree in computer science from the University of Illinois at Urbana-Champaign, in 2005. He is an associate professor with the Computer Engineering Department, Bilkent University. He worked with Intel Corporation between 2005-2015, most recently as a research scientist with the Strategic CAD Labs, Hillsboro, Oregon. He has served in the executive and technical program committees of several conferences. He is a recipient of the IEEE/ACM William J. McCalla ICCAD Best Paper Award (2011), ACM SIGDA Technical Leadership Award (2012), TUBITAK Postdoctoral Reintegration Fellowship (2016), and the European Commission MSCA Individual Fellowship (2016). His research interests include high performance computing, computer-aided design algorithms, and hardware/FPGA accelerators for large-scale problems. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.