Combining Data Duplication and Graph Reordering to Accelerate Parallel Graph Processing

Vignesh Balaji
vigneshb@andrew.cmu.edu
Carnegie Mellon University

Brandon Lucia
blucia@andrew.cmu.edu
Carnegie Mellon University

ABSTRACT

Performance of single-machine, shared memory graph processing is affected by expensive atomic updates and poor cache locality. Data duplication, a popular approach to eliminate atomic updates by creating thread-local copies of shared data, incurs extreme memory overheads due to the large sizes of typical input graphs. Even memory-efficient duplication strategies that exploit the power-law structure common to many graphs (by duplicating only the highly-connected "hub" vertices) suffer from overheads for having to dynamically identify the hub vertices. Degree Sorting, a popular graph reordering technique that re-assigns hub vertices consecutive IDs in a bid to improve spatial locality, is effective for single-threaded graph applications but suffers from increased false sharing in parallel executions.

The main insight of this work is that the combination of data duplication and Degree Sorting eliminates the overheads of each optimization. Degree Sorting improves the efficiency of data duplication by assigning hub vertices consecutive IDs which enables easy identification of the hub vertices. Additionally, duplicating the hub vertex data eliminates false sharing in Degree Sorting since each thread updates its local copy of the hub vertex data. We evaluate this mutually-enabling combination of power-law-specific data duplication and Degree Sorting in a system called RADAR. RADAR improves performance by eliminating atomic updates for hub vertices and improving the cache locality of graph applications, providing speedups of up to 165x (1.88x on average) across different graph applications and input graphs.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Computer systems organization** → **Multicore architectures**.

KEYWORDS

Graph Processing, Atomics, Locality, Power-law, Data Duplication

ACM Reference Format:
Vignesh Balaji and Brandon Lucia. 2019. Combining Data Duplication and Graph Reordering to Accelerate Parallel Graph Processing. In *The 28th International Symposium on High-Performance Parallel and Distributed Computing(HPDC'19), June 22–29, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3307681.3326609

HPDC '19, June 22–29, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6670-0/19/06…$15.00
https://doi.org/10.1145/3307681.3326609

1 INTRODUCTION

Graph processing is an important category of data-intensive workloads with many high-value applications including social network analysis, path-planning, graph learning, data mining, and semi-supervised learning [1, 12, 17]. In the past, large graphs have been processed using distributed systems and datacenter scale systems [14, 19, 26]. More recently, graph processing has seen a shift away from distributed systems as increasing main memory capacities and core counts enable processing graphs with hundreds of millions of vertices and billions of edges using a single machine. Recent work demonstrated that when a graph can fit in a single machine's main memory, distributed graph processing frameworks are less efficient than processing on a single machine [21]. The viability of efficient single-machine graph processing allows computing at the *edge* [29], obviating the need to transmit graph data to a datacenter or to maintain a complex distributed system.

Achieving high performance for parallel graph processing on a single machine requires addressing several unique challenges. Performance characterizations on server-class processors have revealed that graph applications are bottlenecked by poor cache locality and expensive atomic instructions [4, 6, 7, 35]. Graph applications exhibit an irregular memory access pattern that reduces cache locality and requires the use of atomic instructions to ensure correctness in a parallel execution. These atomic instructions impose a significant performance penalty in graph applications [6]. The unique characteristics of graph processing make it challenging to employ optimizations targeting a reduction in expensive main memory accesses and atomic instructions.

One common strategy for eliminating atomic instructions is to create per-thread copies of the shared data (which we refer to as data duplication). Data duplication effectively eliminates the cost of atomics by forcing a thread to apply updates only to its private copy of the vertex data, and periodically combining threads' copies. Data duplication is effective for small data structures with regular access patterns, but incurs high overheads for graph applications. Duplicating the entire vertex data across threads incurs an extreme memory overhead: a graph with 100 Million vertices processed by 32 threads suffers an untenable duplication overhead of 12.8GB, assuming 4 bytes per vertex. The overhead eliminates already scarce locality in the Last Level Cache (LLC), increasing DRAM accesses that degrade performance.

A more memory-efficient implementation of data duplication might exploit the *power-law degree distribution* common to many input graphs. A power-law graph contains a small fraction of "hub"

vertices with disproportionately higher connectivity than most vertices in the graph. A vertex with higher degree is likely to be accessed and updated more frequently than lower degree vertices. Consequently, when processing a power-law graph, a majority of atomics update the hub vertices. We propose that a more efficient data duplication approach should duplicate hubs only, eliminating most atomics while avoiding the memory bloat of full-graph duplication. A lingering inefficiency in such a data duplication implementation is the cost of identifying whether a vertex is a hub or not. Data duplication for power-law graphs suffers from a critical limitation: a hub may reside anywhere in the graph's vertex array, requiring the memory-efficient data duplication variant to pay a cost (to determine whether a vertex is a hub) *on every update*.

An orthogonal approach to optimizing graph computations is to reorder vertex data, ensuring that vertices likely to be accessed together are stored together to expose spatial locality. Degree Sorting assigns vertex IDs in decreasing order of vertex degrees, which consecutively orders hub vertices at the start of the vertex array. Hub vertices are likely to be accessed together because of their high connectivity and Degree Sorting improves spatial locality of hub accesses [3, 34, 37], improving performance. Despite the locality improvement, Degree Sorting suffers from a critical limitation: placing the hub vertices in the same cache line leads to *false sharing* when different processors access different hubs in the same cache line. False sharing increases cache coherence activity and degrades performance.

In this work we observe that there is an important synergy between reordering and duplication optimizations for power-law graph data. *These optimizations are mutually enabling*. Degree Sorting optimizes power-law-specific data duplication because detecting a hub vertex has a negligible cost if the input graph's vertices are sorted by their degree. A vertex is identifiably a hub if it is at the start of the vertex array. Data duplication optimizes Degree Sorting because duplicating hub vertices eliminates the false sharing for hub vertices and its attendant coherence cost and performance degradation.

In this paper, we thoroughly motivate and describe RADAR [1], a system that combines duplication and reordering into a single graph processing optimization, reaping the benefits and eliminating the costs of both. RADAR improves performance of graph applications by reducing the number of atomic updates *and* improving locality of memory accesses, providing speedups of up to 165x (1.88x on average) across widely varied graph applications and power-law input graphs. RADAR is an alternative to the state-of-the-art optimization for eliminating the cost of atomics in graph applications, which is to perform the Push-Pull direction-switching optimization [4, 7, 30]. Push-Pull optimization avoids atomic instructions in graph applications by redundantly processing edges, trading off work-efficiency for a reduction in atomics. RADAR offers three key benefits over push-pull optimization. First, unlike Push-Pull, RADAR *does not compromise work-efficiency* in a graph algorithm. Instead, RADAR leverages power-law graph structure to eliminate atomic updates for hub vertices. Second, RADAR *combines* the benefits of data duplication's reduction in atomics and Degree Sorting's improvement in spatial locality. Third, and often most critically, RADAR requires *half of the memory footprint* of Push-Pull, because RADAR need not maintain an in-memory representation of in-edges and out-edges, as required in Push-Pull. The reduced memory requirement allows RADAR to process a substantially larger input graph than push-pull, on a single machine with a fixed memory capacity.

In summary, we make the following contributions in this work:

- We identify the challenges of implementing memory-efficient data duplication for power-law graphs and (to the best of our knowledge) provide the first implementation of a power-law-specific data duplication strategy for shared-memory graph processing (Section 4)
- We show that by combining the mutually beneficial optimizations of data duplication and Degree Sorting, RADAR achieves speedups of up to 165x (1.88x on average) across different applications and input graphs (Section 7).
- We compare RADAR's performance to the push-pull optimization and identify the scenarios where each optimization provides the best performance.

2 BACKGROUND: GRAPH PROCESSING

Our work is motivated by the performance limitations imposed on shared memory parallel graph processing by atomic updates and poor cache locality. This section provides an overview of shared memory graph processing by describing standard graph data structures and algorithms, as well as common optimizations. While there is diversity in the optimizations employed by different graph processing frameworks [28], shared memory frameworks share similarities in the data structure used to represent graphs, graph traversal patterns, and optimizations commonly used to eliminate atomic accesses.

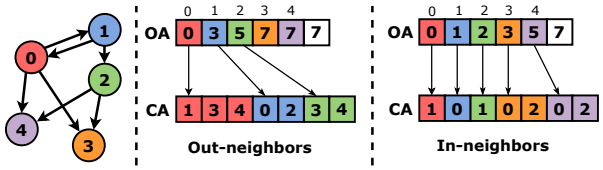


Figure 1: CSR representation of a directed graph

Compressed Sparse Row representation. Most shared memory frameworks represent a graph using the *Compressed Sparse Row* (CSR) format due to the format's memory efficiency [4, 25, 30, 33, 36]. Figure 1 illustrates a graph in the CSR format. The CSR format uses two arrays to represents a graph's edges (sorted by edge source/destination ID). The Coordinates Array (CA) contiguously stores the neighbor of each vertex in the graph. The Offsets Array (OA) stores each vertex's starting offset into the Coordinates Array. To access the neighbors of vertex i, a program accesses the i^{th} entry in OA to find vertex i's first neighbor in the CA. The OA allows quick estimation of vertex degree: vertex i's neighbor count is the difference between entries $i+1$ and i in the OA. To represent a directed graph, an algorithm or framework may store *two* CSRs (e.g., Figure 1), one for *outgoing* neighbors and another for *incoming* neighbors.

Graph Traversal Pattern. Graph applications process an input graph by iteratively visiting its vertices until a convergence criterion is satisfied. During an iteration, an application may either

[1] Due to the mutually enabling combination of Duplication and Reordering, we name our system RADAR (Reordering Assisted Duplication/Duplication Assisted Reordering)

Algorithm 1 Typical graph processing kernel

```
1: par_for src in Frontier do
2:     for dst in out_neigh(src) do
3:         AtomicUpd (vtxData[dst]), auxData[src])
```

process all the vertices or may process a subset of vertices called the *frontier*. Also, within each iteration, vertices belonging to the next iteration's frontier are identified using application-specific logic. Algorithm 1 shows an arbitrary graph processing kernel that traverses an input graph, applying an update to vertex data (`vtxData`) based on auxiliary data (`auxData`). The kernel processes the vertices in a frontier in parallel (line 1) and accesses the neighbors of each vertex in the frontier (line 2). Note, that the neighbors of a vertex are identified using the contents of the Coordinates Array (CA) of the CSR. The update operations applied to elements of `vtxData` are typically associative and commutative.

The kernel illustrates key performance challenges faced by shared memory graph processing frameworks. First, the kernel must use expensive atomic instructions to synchronize each update to an entry in the vertex data array (`vtxData` in line 3). The updates require synchronization because multiple source vertices can share the same destination vertex as a neighbor, causing concurrent updates of shared neighbors. Second, accesses to `vtxData` in line 3 suffer from poor cache locality. The sequence of accesses to the `vtxData` array is determined by the contents of the Coordinates Array (CA) of the CSR. The `vtxData` accesses are unlikely to have spatial or temporal locality because each access is an indirect lookup that depends on graph structure and vertex ordering (both of which can be arbitrary).

Algorithm 2 Pull version of graph kernel

```
1: par_for dst in G do
2:     for src in in_neigh(dst) do
3:         if src in Frontier then
4:             Upd (vtxData[dst]), auxData[src])
```

Push-Pull direction-switching. Shared-memory graph processing frameworks often use an optimized graph traversal pattern called a *pull phase* execution [4, 7, 30] to eliminate atomic updates. The typical graph processing kernel shown in Algorithm 1 can be classified as a *push phase* execution since a vertex is processed by iterating over its outgoing neighbors, "pushing" the value of a vertex to its out-neighbors. In contrast, a *pull phase* execution processes a vertex by iterating over its incoming neighbors, "pulling" the value from all of its in-neighbors (shown in Algorithm 2). The pull phase requires no atomic updates because only one thread updates a vertex. However, the elimination of atomics comes at the cost of processing redundant edges. For example, for the graph shown in Figure 1 if the *Frontier* consists of vertices 0 and 2, then a push phase execution would iterate over outgoing edges from the two vertices and update `vtxData[3]` and `vtxData[4]` using atomic instructions. In contrast, a pull phase execution would iterate over incoming edges of *every* vertex (lines 1 and 2 of Algorithm 2) and check if the source vertex of each incoming edge belongs to the *Frontier* (i.e. whether the source vertex is 0 or 2). When an edge connected to *Frontier* vertices is discovered, a pull phase execution updates `vtxData` without using atomic instructions. However, a pull phase execution eliminates atomics updates by inspecting more edges compared to the push phase execution (7 versus 4 in our example) and, hence, is *work-inefficient*. The pull phase trades off work-efficiency in order to eliminate atomic updates in graph applications and has shown to be effective only when the frontier contains a majority of the vertices (i.e., the frontier is *dense*) [4, 7, 30]. Therefore, graph frameworks employ Push-Pull direction-switching to dynamically switch between push and pull phases only using the pull phase when the frontier is dense, using push otherwise. Consequently, the Push-Pull optimization is required to store two CSRs in memory – one for tracking outgoing neighbors (used in push phase) and another for tracking incoming neighbors (used in pull phase). A critical limitation of the Push-Pull optimization is that it has double the memory footprint compared to a baseline, push phase execution.

3 THE CASE FOR RADAR

We demonstrate the extent to which atomic updates impact the performance of parallel graph applications. We then show the tension between locality improvements from graph reordering (e.g., Degree Sorting) and a commensurate performance degradation due to false sharing in a parallel execution. These costs motivate RADAR, which synergistically combines duplication and reordering to reduce atomic updates and improve cache locality of vertex data accesses.

3.1 Atomics impose significant overheads

Atomic instructions impose a significant penalty on graph applications [6, 35]. Compared to other kinds of graphs, the performance cost of atomic updates is higher while processing power-law graphs because the highly-connected hub vertices are frequently updated in parallel. To motivate RADAR, we experimentally measured the performance impact of atomic updates (Section 6 describes our experimental setup in detail). We compare the performance of a baseline execution with a version that replaces atomic instructions with plain loads and stores. To ensure that the latter version produces the correct result and converges at the same rate as the baseline, we execute each iteration twice: once for timing without atomics, and once for correctness with atomics (but not timed).

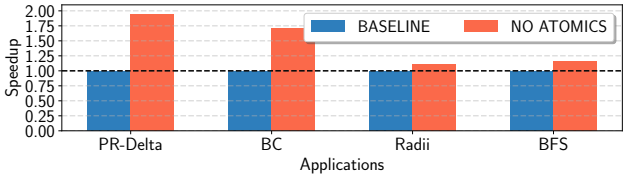


Figure 2: Speedup from removing atomic updates

Figure 2 shows the impact of atomic updates for several applications processing `PLD`, a power-law graph. The data show the performance potential of eliminating atomic updates in different applications. Pagerank-Delta (PR-Delta) and Betweenness-Centrality (BC) see a large improvement, indicating a high cost due to atomics. Breadth-first Search (BFS) and Radius Estimation (Radii) see a smaller improvement. These two algorithms allow applying the Test-and-Test-and-Set optimization [27], which avoids atomics for

already-updated vertices in the baseline. Across the board, the data show the opportunity to improve performance by eliminating atomic updates.

3.2 Data duplication for power-law graphs

Data duplication is a common optimization used in distributed graph processing systems [19], where vertex state is replicated across machines to reduce inter-node communication. Recent distributed graph processing systems [10, 14, 26] have leveraged the power-law degree distribution, common to many real world graphs, to propose data duplication only for the highly connected *hub* vertices. Duplicating only the hub vertex data improves memory efficiency by incurring the overheads of duplication only for the small fraction of hub vertices (that contribute the most to inter-node communication). In this work, we explore data duplication of hub vertices in power-law graphs (henceforth referred to as HUBDUP) in the context of single node, shared memory graph processing to reduce inter-core communication caused by atomic updates. Specifically, we create thread-local copies of hub vertex data which allows threads to independently update their local copy of the vertex data without using atomic instructions. Later, threads use a parallel reduction to combine their copies, producing the correct final result.

The hub-specific duplication strategies proposed in recent distributed graph processing systems [10, 14, 26] cannot be directly applied to the shared memory setting because of fundamental differences in the primary performance bottlenecks. The primary bottleneck in distributed graph processing is expensive inter-node communication [28]. To reduce communication over the network, distributed graph processing systems require sophisticated preprocessing algorithms to effectively partition a graph's edges across nodes *in addition* to duplicating hub vertex data. The high preprocessing costs of these algorithms are harder to justify in the context of shared memory graph processing due to the relatively low cost of communication (between cores within a processor). The primary bottleneck in single node, shared memory graph processing is the latency to access main memory (DRAM) [28, 37]. Therefore, efficient data duplication for shared memory graph processing must ensure that the increased memory footprint from duplication can be serviced from the processor's Last Level Cache (LLC). Otherwise, the performance benefits of eliminating atomic updates would be overshadowed by an increase in DRAM accesses. The limited capacity of typical LLCs (order of MBs) allows shared memory graph processing frameworks to duplicate far fewer vertices than distributed graph processing systems.

Despite the significant overhead imposed by atomic instructions (Figure 2), popular shared memory graph processing frameworks [4, 25, 30, 33] do not use data duplication (including HUBDUP). Achieving high performance from HUBDUP requires careful implementation to avoid excessive overheads. A key challenge facing any HUBDUP implementation is that a hub vertex may initially have an arbitrary position in the vertex data array. A memory-efficient HUBDUP implementation must dynamically identify whether a vertex being updated is a hub or not *at runtime*. Consequently, a HUBDUP implementation will remain sub-optimal; while HUBDUP successfully eliminates atomics for hubs, its incurs a run time cost to identify those hubs.

3.3 Graph reordering for locality

Graph applications are notorious for their poor cache locality as evident from the large body of work focusing on optimizing locality for graph processing [5, 23, 34, 35, 37]. Reordering the vertices in a graph in decreasing degree order (which we refer to as Degree Sorting) is one such optimization that exploits the power-law degree distribution. Degree Sorting causes the vertices with the highest degrees (i.e., hubs) to be assigned at the start of the vertex data array. An access to a hub vertex's data also caches the data for the other hubs in the same cache line. Bringing more hubs into the cache increases the likelihood that future requests to hub vertices hit in the cache, improving performance. Degree Sorting is appealing because it is a preprocessing step that requires *no* modification to application code. While more sophisticated graph ordering techniques exist [8, 15, 34], Degree Sorting has the benefit of having a relatively low preprocessing overhead. The high preprocessing cost of other approaches negates the benefit of reordering [2, 3].

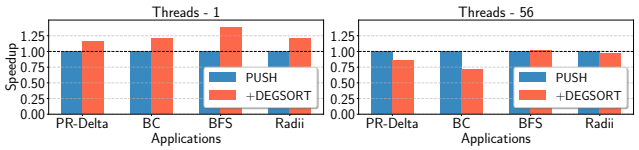


Figure 3: Performance improvements from Degree Sorting: *Reordering improves performance of a single-threaded execution but fails to provide speedups for parallel executions.*

Figure 3 shows the performance improvements offered by reordering vertices in decreasing in-degree order while processing the PLD power-law graph, for different thread counts. In a *single-threaded* execution, Degree Sorting's locality optimization effectively improves performance. However, in a parallel execution with 56 threads, Degree Sorting causes a *slowdown*. False sharing causes the parallel performance degradation, because Degree Sorting lays the most commonly accessed vertices (hubs) consecutively in memory. As threads compete to access the cache lines containing these vertices, they falsely share these lines, suffering the latency overhead of cache coherence activity Single-threaded executions show that Degree Sorting is effective, but in a highly parallel execution the cost of false sharing exceeds the benefit of Degree Sorting, leading to performance loss.

3.4 Benefits of combining duplication and reordering

The previous sections show that optimizations targeting a reduction in atomic updates and improvement in cache locality both have limitations. HUBDUP reduces atomic updates but incurs a cost to detect the hub vertices at runtime. Degree Sorting improves cache locality for a single-threaded execution, but suffers from false sharing in a parallel execution.

Our main insight in this work is that combining the two optimizations — i.e., HUBDUP on a degree-sorted graph — alleviates the bottleneck inherent in each. Reordering the input graph in decreasing degree-order locates hubs contiguously in the vertex array, enabling a HUBDUP implementation to identify the hub vertices at

a lower cost. Duplicating vertex data for hubs eliminates the false-sharing incurred by Degree Sorting, because each thread updates a thread-local copy of hub vertex data. Table 1 shows an overview of existing techniques with their strengths and weaknesses, including RADAR, the technique we develop in this work.

Optimization	Summary	Strengths/Weaknesses
HUBDUP	Duplicating only hub data (in original graph order)	+ No atomics for hub vertices - Cost for identifying the hubs
Degree Sorting	Reorder graph in decreasing degree order (no app change)	+ Improves cache locality - Introduces false sharing
RADAR	Duplicating only hub data on a degree sorted graph	+ No atomics for hubs (with easy hub detection) + Improves cache locality (no false sharing on hub updates)

Table 1: Summary of optimizations

4 RADAR: COMBINING DUPLICATION AND REORDERING

RADAR combines the mutually-beneficial HUBDUP and Degree Sorting optimizations, providing better performance compared to applying either optimization in isolation. To motivate RADAR's design, we first describe the space of HUBDUP designs, characterizing the fundamental costs associated with any HUBDUP implementation. We then discuss how Degree Sorting reduces the inefficiencies of HUBDUP. Finally, we discuss how RADAR combines reduction in atomic updates with improvements in cache locality to improve performance of graph applications.

4.1 Sources of inefficiency in HUBDUP

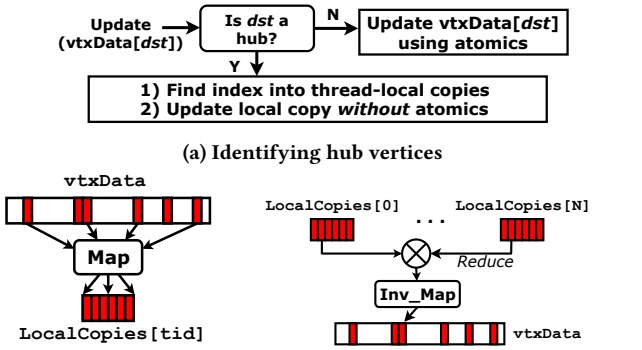
Power-law graphs present an opportunity to develop memory-efficient data duplication implementations (HUBDUP). Despite the low memory overhead of duplicating hub vertex data only, HUBDUP's performance is sub-optimal. Specifically, implementing HUBDUP requires addressing four key challenges.

Challenge #1: Locating hub vertices. Hub vertex data may be arbitrarily located in the vertex data array, because HUBDUP makes no assumption about input graph ordering. A HUBDUP implementation must identify whether a vertex is a hub. One possible implementation is to inspect the entire graph in advance and store an index (i.e., a bitvector) of hub vertex locations.

Challenge #2: Detecting hub vertices. HUBDUP limits memory overheads by duplicating only hub vertex data. A HUBDUP implementation must dynamically check whether a vertex is a hub or not; hub updates modify a thread-local copy, while non-hub updates atomically update the `vtxData` array. An implementation can use the bitvector mentioned above to efficiently make this hub check on every vertex update at run time, as illustrated in Figure 4a

Challenge #3: Updating the thread-local hub copies. Hub updates in HUBDUP are applied to a thread-local copy of the hub's data and do not use atomic instructions. A memory efficient HUBDUP implementation must store hub duplicates contiguously in memory (e.g., `LocalCopies` in Figure 4b). However, packing hub vertices in thread-local copies precludes using a hub vertex's ID as an index to the thread-local copies. HUBDUP requires mapping a hub vertex's ID to its index in the thread-local copies (as in Figure 4b). The mapping function must be called *on every hub update* as shown in Figure 4a.

Challenge #4: Reducing updated hub copies. At the end of an iteration, partial hub updates accumulated in thread-local copies



(a) Identifying hub vertices

(b) Gathering hub vertex updates in thread-local copies

(c) Reduction of values in thread-local copies

Figure 4: HUBDUP design: *Essential parts of any HUBDUP implementation. Hub vertices of the graph are highlighted in red.*

must be reduced and stored in the hub's entry in the `vtxData` array (Figure 4c). Locating a hub copy's original location in the vertex array requires an inverse mapping from its index in the array of thread-local copies back to its index in the original vertex data array.

A key motivation for RADAR is achieving the benefits of HUBDUP without incurring the costs of the above challenges.

4.2 Degree Sorting improves HUBDUP

Degree Sorting improves HUBDUP by reducing the costs associated with each challenge. Most of the cost of HUBDUP stems from the arbitrary location of hubs in the vertex data array. Degree Sorting solves this problem by arranging vertices in decreasing in-degree order. A degree-sorted graph avoids the first two challenges: identifying a hub requires simply checking that its index is lower than a constant threshold index marking the boundary of hub vertices. Indexing thread-local copies is also simple because hubs are contiguous. A hub's vertex ID can be used to directly index into the thread-local copies. Therefore, Degree Sorting eliminates the cost of building and accessing the maps and inverse maps from challenges #3 and #4.

4.3 HUBDUP improves Degree Sorting

Degree Sorting improves cache locality by tightly packing high-degree hubs in the vertex data array. Figure 3 demonstrates the benefit of locality improvements for single-threaded graph applications. However, contiguity of hubs causes an unnecessary increase in costly false sharing because different threads frequently read and update the small subset of cache lines containing hubs. Thread-local hub copies in HUBDUP avoid false sharing because a thread's hub updates remain local until the end of an iteration.

4.4 RADAR = HUBDUP + Degree Sorting

RADAR combines the best of HUBDUP and Degree Sorting by duplicating hub vertex data in a degree sorted graph. Duplication mitigates false sharing and degree sorting keeps the overhead of duplication low. The key motivation behind RADAR is the observation that HUBDUP and Degree Sorting are mutually enabling. The

key contribution of RADAR is the tandem implementation of these techniques, which realizes their mutual benefits.

5 RADAR DESIGN AND IMPLEMENTATION

This section describes the design, implementation, and optimization of RADAR. We begin by first describing the design of an efficient HUBDUP baseline implementation, which does not exist in the literature to the best of our knowledge.

5.1 HUBDUP design decisions

We designed and implemented HUBDUP with the aim of keeping the space and time overheads of duplication low. To identify hubs (challenge #1) we collect the in-degrees of all vertices and then sort the in-degrees to find the threshold degree value for a vertex to be classified as a hub. We use GCC's `__gnu_parallel::sort` to sort indices by degree efficiently. Note that sorting indices is much simpler than re-ordering the graph according to the sorted order of indices (Degree Sorting). We use the `hMask` bitvector to dynamically detect hubs (i.e., challenge #2), setting a vertex's bit if its degree is above a threshold. `hMask` has a low memory cost: a 64M-vertex graph requires an 8MB bitvector to track hubs, which is likely to fit in the Last Level Cache (LLC). We use an array to implement the mapping from a hub's vertex ID to its index into the thread-local copies and another array for the inverse mapping. HUBDUP populates both arrays in advance.

5.2 Optimizing reduction costs

After each iteration, both HUBDUP (and RADAR) must reduce updated thread-local hub copies and store each hub's reduction result in its entry in the vertex array. Only a subset of hub vertices may need to be reduced in a given iteration because frontier-based applications update the neighbors of vertices in the frontier only (Algorithm 1). An efficient implementation of HUBDUP should avoid reducing and updating hubs that were not updated during that iteration. HUBDUP and RADAR explicitly track an iteration's updated hub vertices with a `visited` array that has an entry for each hub. An update to a hub (by any thread) sets the corresponding entry in the `visited` using the Test-and-Test-and-Set operation (if a hub's `visited` bit is set once in an iteration, it is never set again until the next iteration). After an iteration, HUBDUP reduces the thread-local, partial updates of each hub that has its `visited` entry set and does nothing for other hubs. This `visited` optimization to reduction improved RADAR performance by up to 1.25x (geometric mean speedup of 1.02x).

5.3 Selecting hub vertices for duplication

Due to limited Last Level Cache (LLC) capacity, duplicating all the hubs [2] in a graph may impose excessive memory overheads. Instead of duplicating all hubs, HUBDUP and RADAR should duplicate only a subset of hubs (hub vertices with the highest degrees) such that the sum of the sizes of all threads' duplicates is less than the capacity of the LLC (Last Level Cache). We demonstrate the importance of duplicating only a subset of hubs by comparing our LLC-capacity-guided duplication strategy ("CACHE-RADAR") to a variant that duplicated all hubs ("ALL-HUBS-RADAR"). Figure 5 shows their relative performance running all applications on `DBP`, the smallest graph in our dataset. The data show that CACHE-RADAR consistently outperforms ALL-HUBS-RADAR, with better LLC locality for all threads' hub duplicates. The performance gap is likely to grow with graph size, as more hub duplicates compete for fixed LLC space. Our CACHE-RADAR design most effectively uses the scarce LLC space to keep only the highest-degree hubs' duplicates cached.

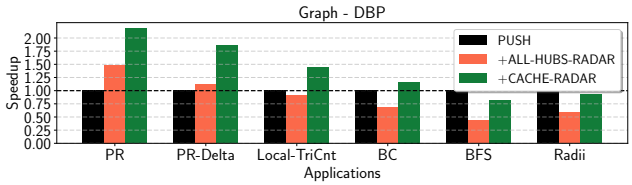


Figure 5: Performance of RADAR with different amounts of duplicated data: *The duplication overhead of ALL-HUBS-RADAR are significant even for the smallest input graph.*

$$numHubs = \frac{S * LLC\_Size}{(T * elemSz) + \delta} \quad (1)$$

This result demonstrates the importance of calibrating HUBDUP and RADAR to the properties of the machine (LLC size). We use Equation 1 to calculate the number of hubs to duplicate in HUBDUP and RADAR. S is a scaling factor (between 0 and 1) that controls the fraction of LLC to be reserved for storing duplicated data, T is the number of threads used, and *elemSz* is the size (in bytes) of each element in the `vtxData` array. The δ parameter in the denominator of Equation 1 accounts for memory overheads in HUBDUP and RADAR. Both RADAR and HUBDUP use a `visited` boolean array to optimize reduction (Section 5.2). We set $\delta = 1$, because the `visited` array has an entry for each hub. For HUBDUP, maintaining the `hMask` bitvector and maps to and from a hub's duplicate's location are additional memory costs and we set $\delta = 3$. On varying the S parameter, we empirically determined that S = 0.9 often provided the best performance across applications and graphs.

6 EXPERIMENTAL SETUP

We now describe the evaluation methodology used for our quantitative studies. We evaluate the performance improvement from RADAR on a diverse set of applications running on large real-world input graphs that stress the limits of memory available in our server.

6.1 Evaluation platform and methodology

We performed all of our experiments on a dual-socket server machine with two Intel Xeon E5-2660v4 processors. Each processor has 14 cores, with two hardware threads each, amounting to a total of 56 hardware execution contexts. Each processor has a 35MB Last Level Cache (LLC) and the server has 64GB of DRAM provided by eight DIMMs. All experiments were run using 56 threads and we pinned the software thread to hardware threads to avoid performance variations due to OS thread scheduling. To further reduce sources of performance variation, we also disabled the "turbo boost" DVFS features and ran all cores at the nominal frequency of 2GHz.

[2] As in prior work [37], we define a hub as a vertex with above-average degree.

We ran 4 trials for the long-running applications (Pagerank and Local Triangle Counting) in our evaluation set, and 11 trials for all the other applications. While computing the mean time, we exclude the timing for the first trial to allow processor caches to warm up. For the source-dependent BFS application, we select a source vertex belonging to the largest connected component in the graph. We maintain a mapping between vertex IDs before and after reordering to ensure that source-dependent applications running on the reordered graphs use the same source as a baseline execution on the original graph [4].

6.2 Applications

We evaluate the performance of RADAR [3] across five applications from the Ligra benchmark suite [30] and one application from the GAP [4] benchmark suite. All the applications were compiled using g++-6.3 with -O3 optimization level and use OpenMP for parallelization. We provide a brief description of the execution characteristics of each application, identifying the `vtxData` array and atomic update operation performed on `vtxData`.

Pagerank (PR): Pagerank is a popular graph benchmark that iteratively refines per-vertex ranks (`vtxData`) until the sum of all ranks drops below a convergence threshold. The application processes all the vertices in a graph every iteration and, hence, performs many random writes to the `vtxData` array. PR uses atomic instructions to increment vertex ranks of destination vertices based on properties of neighboring source vertices.

Pagerank-delta (PR-Delta): Pagerank-delta is a variant of Pagerank that does not process all the vertices of a graph each iteration. Instead, PR-Delta only processes a subset of vertices for which the rank value changed beyond a δ amount, which improves convergence [19]. Even though PR-Delta does not process all vertices every iteration, the application processes dense frontiers during the initial iterations of the computation which generate many random writes to the `vtxData` array. PR-Delta uses atomic instructions in a similar fashion to PR.

Betweenness Centrality (BC): Betweenness-Centrality iteratively executes a BFS kernel from multiple sources to count the number of shortest paths passing through each vertex (`vtxData`). Most iterations of BC process sparse frontiers (i.e. frontier contains a small fraction of total vertices). BC also performs a transpose operation (exchanging incoming neighbors with outgoing neighbors and vice versa) and, hence, is an application that needs to store two CSRs even in a baseline push-based execution. BC uses atomic instructions to increment the number of shortest paths passing through each vertex.

Radii Estimation (Radii): Graph Radii estimation approximates the diameter of a graph (longest shortest path) by performing simultaneous BFS traversals from many randomly-selected sources. Radii uses a bitvector (`vtxData`) to store information about BFS traversals from multiple source vertices. Atomic instructions are used to atomically perform a bitwise-OR on the `vtxData` array. Unlike applications discussed so far, subsequent updates to the `vtxData` array might produce no change to the `vtxData` array. Therefore, Radii uses the Test-and-Test-and-Set (T&T&S) optimization [27] to avoid executing atomic instructions for updates that will produce no change.

Breadth First Search (BFS): BFS is an important graph processing kernel that is often used as a subroutine in other graph algorithms. The kernel iteratively visits all the neighbors reachable from a particular source, identifying a parent for each vertex (`vtxData`). Similar to Radii, BFS also uses the T&T&S optimization to atomically set a parent for each vertex.

Local Triangle Counting (Local-TriCnt): Local Triangle Counting identifies the number of triangles (or cliques of size 3) incident at each vertex (`vtxData`) of an undirected graph and is a variant of the Triangle Counting benchmark that only reports the total count of triangles. Our Local-TriCnt implementation extends the optimized Triangle Counting implementation from GAP which performs Degree Sorting on the input graph to achieve an algorithmic reduction in the number of edges to be processed. Finding the number of triangles per-vertex allows computing a graph's local clustering coefficients which has applications in identifying tightly-knit communities in social networks [12]. Local-TriCnt uses atomic instructions to increment the number of triangle discovered for each vertex.

	DBP	GPL	PLD	TWIT	MPI	KRON	WEB	SD1
Reference	[16]	[13]	[22]	[17]	[16]	[4]	[11]	[22]
$\|V\|$ (in M)	18.27	28.94	42.89	61.58	52.58	67.11	50.64	94.95
$\|E\|$ (in B)	0.172	0.462	0.623	1.468	1.963	2.103	1.93	1.937
Avg. Degree	9.4	16	14.5	23.8	37.3	31.3	38.1	20.4
% of Hubs	11.75	20.54	14.72	11.30	9.52	8.43	5.56	10.61

Table 2: Statistics for the evaluated input graphs

6.3 Input graphs

To evaluate RADAR's performance, we use large, real-world, power-law input graphs that have been commonly used in other academic research. We also include the kronecker synthetic graph in our evaluation due to their popularity in the graph500 community [24]. Table 2 lists key statistics for the graphs in our dataset. Unless noted otherwise, we use the original vertex data ordering of the input graph as provided by the authors of the graph dataset for our baseline executions.

7 EVALUATION

RADAR combines the benefits of HUBDUP and Degree Sorting to improve performance of graph applications. We evaluate the performance of RADAR using applications and input graphs described in Section 6. We also compare RADAR's performance to the Push-Pull direction-switching optimization and identify scenarios where each optimization offers superior performance.

7.1 Performance analysis of RADAR

To illustrate the benefits of combining HUBDUP and Degree Sorting, we compare RADAR's performance with executions that either perform HUBDUP or Degree Sorting. Figure 6 shows the performance of HUBDUP, Degree Sorting, and RADAR relative to a baseline, push-style execution of graph applications across 8 input graphs. We report the key findings from the results below:

Finding 1: RADAR outperforms HUBDUP and Degree Sorting. For PR, PR-Delta, Local-TriCnt, and BC, RADAR consistently provides higher speedups than only performing HUBDUP or Degree Sorting. The results highlight the synergy between HUBDUP and

[3] Source code for RADAR (and all the other optimizations) is available at `https://github.com/CMUAbstract/RADAR-Graph`

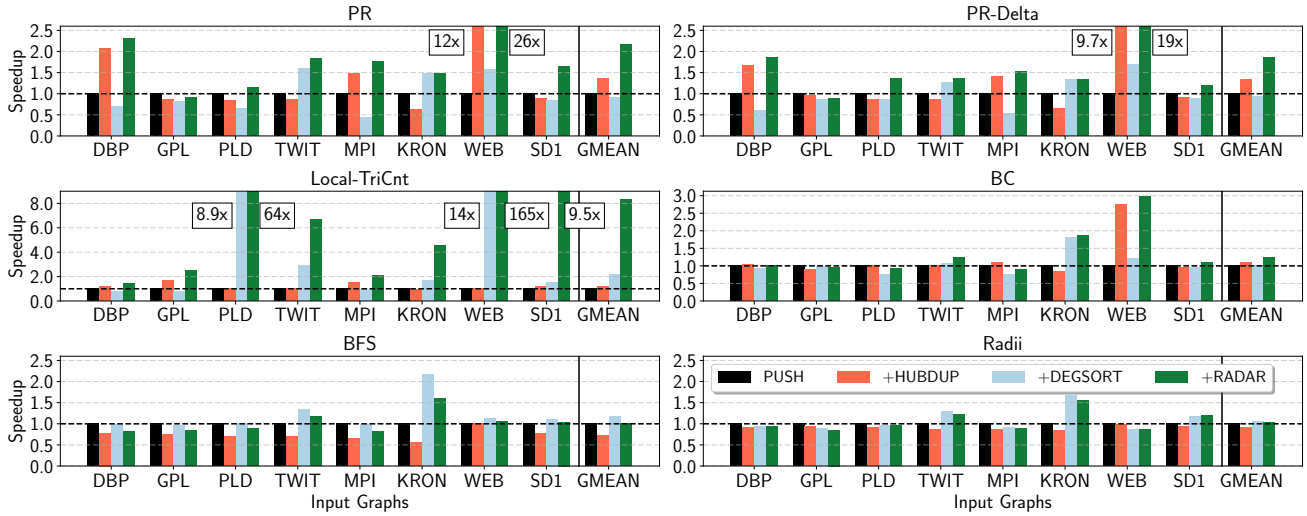


Figure 6: Comparison of RADAR to HUBDUP and Degree Sorting: *RADAR combines the benefits of HUBDUP and Degree Sorting, providing higher speedups than HUBDUP and Degree Sorting applied in isolation.*

Degree Sorting which is exploited by RADAR to provide additive performance gains.

Finding 2: HUBDUP offers speedup in graphs with good ordering of hubs. For the four applications mentioned above, HUBDUP only provides speedups on three input graphs - DBP, MPI, and WEB - often causing a slowdown for the other input graphs. HUBDUP performs well in the DBP, MPI, and WEB graphs because most hub vertices are consecutively ordered in these graphs. Due to the consecutive ordering of hub vertices, accesses to key HUBDUP data structures – the hMask bitvector and the mapping from hub vertex ID to locations in thread-local copies – benefit from improved locality, driving down the costs of an HUBDUP execution.

	DBP	GPL	PLD	TWIT	MPI	KRON	WEB	SD1
PR Speedup	2.06x	0.85x	0.84x	0.86x	1.49x	0.62x	11.91x	0.90x
Unique words	17.03K	22.39K	64.32K	68.80K	13.03K	71.38K	6.02K	53.56

Table 3: Number of unique words in the hMask bitvector containing hub vertices: *HUBDUP offers the highest speedups for graphs in which hubs map to the fewest number of unique words in the bitvector.*

Table 3 demonstrates the relation between HUBDUP performance and the vertex order of the graph by showing speedup from HUBDUP for PR along with the number of unique words in the hMask bitvector corresponding to hub vertices. For the same number of hubs (a machine-specific property as described in Section 5.3), hubs in the DBP, MPI, and WEB graphs map to fewer words in the hMask bitvector (an 8B word in the bitvector encodes information for 64 vertices). Fewer words associated with hub vertices improves locality of hMask accesses and allows HUBDUP to provide speedups. The results indicates that HUBDUP is likely to provide speedups for input graph orderings where hub vertices have nearly consecutive IDs.

Finding 3: Degree Sorting can lead to performance degradation. Degree Sorting causes slowdowns in PR, PR-Delta, and BC for certain input graphs. Section 3.3 showed that the performance degradation is due to increased coherence traffic caused by false sharing between threads updating hub vertex data. The slowdown from increased coherence traffic is higher for applications that update more vertices each iteration (PR and PR-Delta) because these applications have a higher likelihood of simultaneously updating hub vertices from different threads. RADAR provides speedups in these applications by duplicating hub vertex data to avoid an increase in coherence traffic.

Finding 4: Degree Sorting provides significant speedups for Local-TriCnt. The results for the PLD, TWIT, and WEB graphs on Local-TriCnt show high speedups from Degree Sorting. The high speedups are due to an algorithmic reduction in the number of edges that need to be processed to identify all the triangles in the graph. The GAP implementation of Triangle Counting already uses Degree Sorting. We normalize data to a baseline without Degree Sorting to show the algorithmic improvement from reordering. RADAR provides additional speedups over the algorithmic improvements from Degree Sorting by eliminating atomic instructions, improving performance by 4.7x on average over Degree Sorting.

Finding 5: Degree Sorting performs best for BFS and Radii. In contrast to Finding 1, the results for BFS and Radii show that Degree Sorting consistently outperforms RADAR and HUBDUP. BFS and Radii use the Test-and-Test-and-Set (T&T&S) optimization, which reduces the cost of atomic updates for these applications (Figure 2). The T&T&S optimization also helps avoid an increase in coherence traffic from Degree Sorting, thereby improving locality. For BFS and Radii, the small improvements from eliminating atomic updates do not justify the cost of duplicating hub vertex data causing RADAR to provide lower performance that Degree Sorting.

To demonstrate the relation between reduced speedup from RADAR and the T&T&S optimization, we studied a BFS implementation that does not use T&T&S. Figure 8 shows the performance of Degree Sorting and RADAR for such a BFS implementation. In

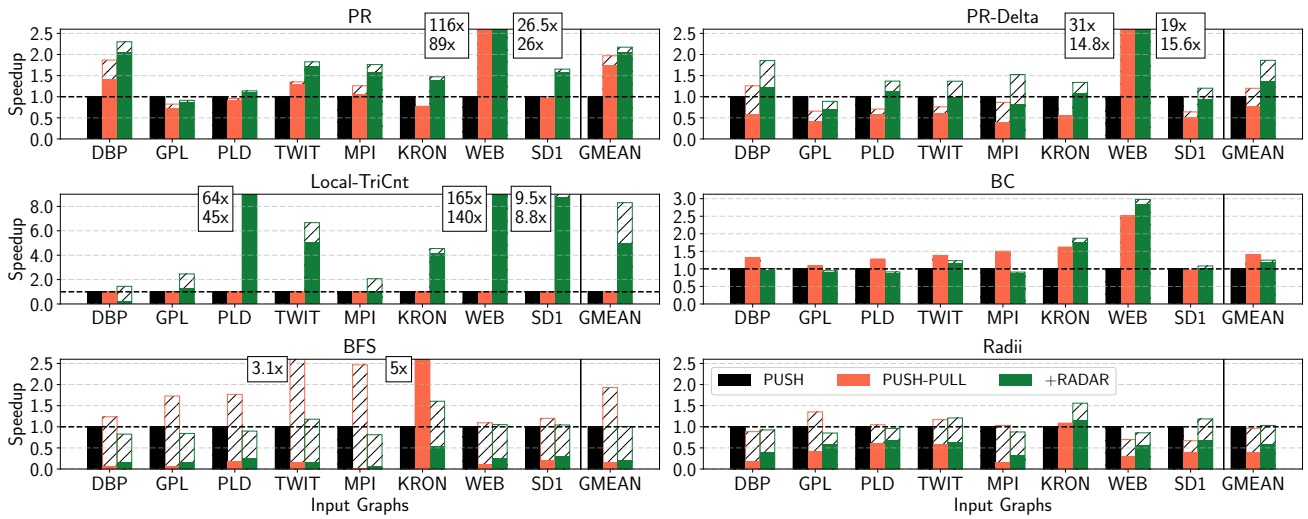


Figure 7: Speedups from Push-Pull and RADAR: *The total bar height represents speedup* ***without*** *accounting for the preprocessing costs of Push-Pull and RADAR. The filled, lower bar segment shows the* ***net speedup*** *after accounting for the preprocessing overhead of each optimization. The upper, hashed part of the bar represents the* ***speedup loss*** *as a result of accounting the preprocessing overhead of each optimization.*

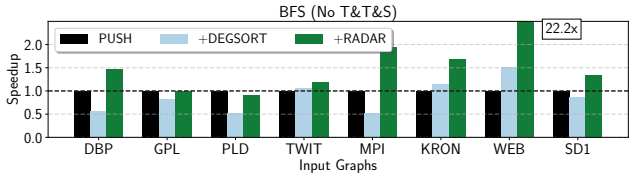


Figure 8: Performance improvements for BFS without the T&T&S optimization: *In the absence of the T&T&S optimization, RADAR outperforms Degree Sorting.*

the absence of T&T&S, the performance results for BFS mirror the results for PR and PR-Delta applications, showing slowdowns with Degree Sorting and speedups with RADAR. While BFS and Radii would never be implemented without the T&T&S optimization, this experiment was useful for showing that RADAR is most effective for applications that cannot use T&T&S.

7.2 Comparison to Push-Pull

We compare RADAR's performance against Push-Pull direction switching which is the state-of-the-art in eliminating atomic updates in graph applications. Figure 7 shows the speedups from RADAR and Push-Pull relative to a baseline, push-based execution. The results show speedups both including the preprocessing overheads (solid bars) and without including the cost of preprocessing (total bar height). In this section, we explain the performance of Push-Pull and RADAR without considering preprocessing overheads (i.e., here we focus on the total bar height) and defer the discussion on performance with preprocessing costs to Section 7.4. The relative benefit of RADAR over Push-Pull is application-dependent and we report our findings for each application below.

Pagerank: Pagerank updates every vertex each iteration (i.e. frontier includes all the vertices). Therefore, every iteration uses the pull phase (Algorithm 2) to eliminate atomic updates. The results show that executing PR using the pull phase improves performance by eliminating atomic updates. However, RADAR outperforms the pull-phase execution because RADAR couples eliminating atomic updates for hub vertices with improved locality. The only exception is the WEB graph which receives significantly higher speedup from Push-Pull because the creators of the graph use a sophisticated algorithm [8], that optimizes pull phase execution, to pre-order the graph at great computational cost [2].

Pagerank-delta: Pagerank-delta uses the Push-Pull optimization to process the graph in the pull phase during dense frontiers (i.e. frontier contains most of the vertices in the graph) and in the push phase otherwise. A pull phase execution eliminates all atomic updates at the expense of reducing work-efficiency. Additionally, a pull phase execution also converts the regular accesses to the *Frontier* in the push phase (line 1 in Algorithm 1) into irregular accesses (line 3 in Algorithm 2). For many graphs, the performance loss from irregular *Frontier* accesses offsets the benefits from eliminating atomic updates and a Push-Pull execution causes slowdowns. In contrast, RADAR provides better performance by eliminating a large fraction of atomic updates while maintaining regular accesses to the *Frontier*. As before, the WEB graph is an exception, where the pull-optimized layout of the graph ensures good locality for *Frontier* accesses. Consequently, Push-Pull eliminates atomic updates without incurring a penalty for *Frontier* accesses, leading to significant performance improvement for the WEB graph.

We explored the trade-off of a pull phase execution, which is to eliminate atomic updates at the expense of making *Frontier* accesses irregular, by running PR-Delta on the same graphs but with two different vertex orders – random ordering and a pull phase optimized ordering called frequency based clustering [37]. Figure 9 shows the speedups from Push-Pull and RADAR on graphs with the two different orderings. Results for the randomly-ordered graphs

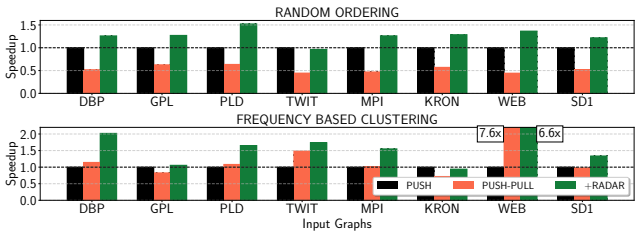


Figure 9: Speedups for PR-Delta from Push-Pull and RADAR on graphs with different orderings: *Push-Pull causes consistent slowdowns when running on randomly ordered graphs.*

show that Push-Pull *consistently* causes slowdowns because the random ordering makes *Frontier* accesses highly irregular, thereby offsetting any performance benefits from eliminating atomics. Note that Push-Pull causes slowdown even in the WEB graph when it is randomly ordered. Push-pull performs better for the graphs ordered with the pull-phase optimized frequency based clustering algorithm. In contrast to Push-Pull, RADAR improves performance *regardless* of the original vertex order of the graph and, hence, is more generally applicable.

Local Triangle Counting: Local Triangle Counting operates on undirected ("symmetrized") versions of input graphs and performs the same accesses in both push and pull phases. For applications like Local Triangle Counting that operate on undirected graphs, RADAR is the only option for improving performance by eliminating atomic instructions.

Algorithm 3 Pseudocode for push-phase of BC

```
1: par_for src in Frontier do
2:     for dst in out_neigh(src) do
3:         if Visited[dst] is True then
4:             AtomicUpd (vtxData[dst]), auxData[src])
```

Betweenness-Centrality: Just like Pagerank-delta, BC uses the Push-Pull optimization by processing dense frontiers using the pull phase and otherwise using the push phase. However, the per-edge computation performed in BC (shown in Algorithm 3) is different from PR-Delta. BC performs an additional check on a *Visited* data structure (line 3) before performing its per-edge computation. For each edge, BC accesses two data structures, each indexed by the source and destination IDs of the edge. Regardless of whether an iteration is processed using the push phase or the pull phase, BC performs irregular accesses to one of the data structures (*Visited* during the push phase and *Frontier* during the pull phase). Therefore, unlike PR-Delta, a pull-phase iteration in BC eliminates atomic updates without introducing irregular accesses. As a result, Push-Pull often outperforms RADAR for BC.

BFS: The Push-Pull direction-switching optimization was originally designed for BFS [4, 30]. In BFS, processing a dense frontier in the pull-phase allows breaking out of the iteration sooner than a push-phase execution. Therefore, Push-Pull significantly outperforms RADAR by achieving an algorithmic reduction in the total number of edges required to be processed.

Radii: Radii has an access pattern similar to Pagerank-delta. However, unlike Pagerank-delta, Radii is not bottlenecked by atomic updates thanks to T&T&S. With little potential for performance improvement from eliminating atomic updates (Figure 2), Push-Pull provides low speedup for Radii.

7.3 RADAR avoids high memory overhead

The Push-Pull optimization doubles an application's memory footprint. Push-Pull implementations switch between executing a graph using push and pull phases based on frontier density and, hence, require two CSRs - one for outgoing neighbors (used during the push phase) and another for incoming neighbors (used during the pull phase). The higher memory footprint of Push-Pull cuts in half the maximum graph size that can be processed using a single machine.

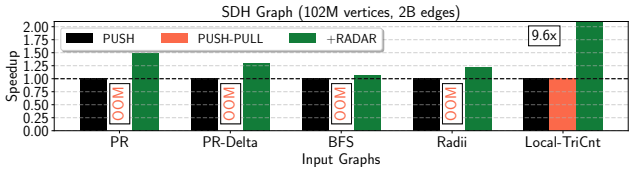


Figure 10: Speedups from RADAR for the SDH graph: *RADAR provides speedups while the size of SDH graph precludes applying the Push-Pull optimization.*

To illustrate the limitations imposed by the higher memory footprint of Push-Pull, we ran experiments on the subdomain host [22] (SDH) graph which is even larger than the SD1 graph. The size of the SDH graph causes an Out of Memory (OOM) error in our 64GB server when storing both the in- and out- CSRs of the graph, making it impossible to apply the Push-Pull optimization for the graph. In contrast, our server fits just the out-CSR of the graph, accommodating the baseline and RADAR versions of applications. Figure 10 shows the performance of RADAR on the SDH graph across different applications. We were unable to run any configuration for BC since even the baseline execution of BC requires both the in- and out-CSRs. The result shows that RADAR provides significant performance improvements for the SDH graph, while Push-Pull runs out of memory. By maintaining the same memory footprint as the baseline, RADAR provides performance improvements for larger graphs than Push-Pull, making RADAR a more effective optimization for graph processing in a single, memory-limited machine.

7.4 Preprocessing overheads

All the optimizations covered in the paper – HUBDUP, Degree Sorting, RADAR, and Push-Pull– require some form of preprocessing on the input graph. The input graph used in Ligra is an adjacency file that stores outgoing edges of the graph in the CSR format. The preprocessing step for Push-Pull builds an in-CSR (CSR for incoming edges) by traversing the input graph to first collect all the incoming edges of the graph and then sorting all the incoming edges by destination IDs[4]. For HUBDUP, the preprocessing step involves populating the `hMask` bitvector for identifying hubs and creating maps between hub vertex IDs and unique locations in thread-local copies of hub data. For RADAR and Degree Sorting, the input graph needs to reordered in decreasing order of degrees.

[4]Ligra uses radix sort to construct the in-CSR.

Reordering the graph requires sorting the in-degrees of vertices to create a mapping from original vertex IDs to new IDs (ordered by decreasing in-degrees) followed by populating a new out-CSR with vertices in the new order. Table 4 lists the preprocessing algorithm complexity and runtime for the different optimizations on all input graphs. HUBDUP has the lowest complexity because it only scans the degrees of all vertices. Push-Pull incurs the maximum complexity because it sorts all the edges to build an in-CSR. Push-Pull, however, incurs zero preprocessing cost for undirected graphs because these graphs have the same incoming and outgoing edges. Finally, RADAR imposes lower preprocessing overhead than Push-Pull for directed graphs.

	Complexity	DBP	GPL	PLD	TWIT	MPI	KRON	WEB	SD1
HUBDUP	$O(V)$	0.06s	0.11s	0.14s	0.24s	0.22s	0.23s	0.19s	0.37s
DegSort/RADAR	$O(VlogV + E)$	0.88s	2.37s	2.29s	8.26s	19.06s	2.94s	3.49s	7.42s
Push-Pull	$O(ElogE)$	2.96s	7.03s	3.91s	9.68s	47.71s	0s	10.86s	12.51s

Table 4: Preprocessing costs for HUBDUP, RADAR, and Push-Pull: *Degree Sorting and RADAR have a smaller preprocessing cost compared to Push-Pull. (V - #vertices and E - #edges)*

Figure 7 shows the speedups from Push-Pull and RADAR after accounting for the above preprocessing costs (filled, lower bar segments). Preprocessing overheads are easily justified for long-running applications such as Pagerank and Local Triangle Counting where RADAR provides a net speedup even after including the preprocessing costs. Preprocessing imposes significant overheads for PR-Delta and Radii. However, these applications are refinement-based algorithms where preprocessing can be justified when more refined results are desired (for example, lower convergence threshold in PR-Delta, traversals from more sources in Radii and BC). Finally, preprocessing overheads can be justified for BFS in scenarios where multiple traversals on the same graph are performed.

8 RELATED WORK

We divide the prior work in graph processing related to RADAR into three categories – data duplication, reducing cost of atomic updates, and locality optimizations.

Data duplication in graph applications: Prior work in distributed graph processing has proposed data duplication for hub vertices in power-law graphs [14, 26]. Vertex delegates [26] replicates hub vertex data across all nodes in a cluster and uses asynchronous broadcasts and reductions to reduce total communication for updating hub data. Powergraph [14] creates replicas of hub vertices to create balanced vertex cuts (assigning equivalent number of edges to each node). As discussed in Section 3.2, the duplication strategies used in the above systems are not directly applicable in the shared memory setting due to differences in the primary bottlenecks of the two scenarios. Data duplication has also been used for reducing the cost of atomic updates in GPU-based graph processing [20]. Similar to RADAR, Garaph highlights the importance of restricting duplication overheads to avoid an increase in DRAM accesses. However, RADAR and Garaph use different techniques to reduce duplication overhead. RADAR creates a per-thread copy only for hub vertices whereas Garaph duplicates all vertices but creates fewer copies than number of threads. The duplication strategy of Garaph is tied to the out-of-core [18] execution model that targets graphs that cannot fit in memory and, hence, is orthogonal to RADAR (which targets in-memory graph processing).

Reducing cost of atomic updates: Prior work has proposed techniques to reduce the cost of atomic updates in graph processing. AAM [6] is a system that uses Hardware Transactional Memory (HTM) to reduce the cost of atomic updates. AAM amortizes the cost of providing atomicity by applying updates for multiple vertices within a single transaction instead of atomically updating each vertex. However, the authors report that AAM is only effective for applications that employ the T&T&S optimization (particularly BFS) and leads to many transaction aborts for applications that cannot employ T&T&S. Therefore, AAM and RADAR are complimentary optimizations because RADAR provides the best performance for applications that cannot employ T&T&S. Galois [25] uses speculative parallelism to avoid fine grained synchronization and improve locality in irregular applications. RADAR also aims to avoid fine grain synchronization and improve locality, but uses data duplication to achieve the goal. Finally, Besta et. al. [7] proposed the "Partition Awareness (PA)" optimization for reducing atomic updates in push-based graph applications. The PA optimization creates two CSRs – one identifying neighbors local to a core and another identifying neighbors belonging to remote cores – allowing threads to update local neighbors without atomic instructions. However, PA requires static partitioning of vertices to thread and precludes dynamic load balancing (which is critical for power-law graphs).

Locality optimizations for graph processing: Extensive research in graph reordering has produced reordering techniques with varying levels of sophistication to improve graph locality [2, 8, 15, 34, 37]. While RADAR could potentially be applied with different reordering techniques, efficient duplication of hub vertices requires that the reordering mechanism produce a graph where the hub vertices are assigned consecutive IDs (Section 4.1). We chose Degree Sorting in our study because of its low overhead and the advantage of assigning hub vertices consecutive IDs at the start of the vertex array. Studying combinations of RADAR with different reordering techniques is an interesting line of research and we leave this exploration for future work.

Vertex scheduling is an alternative to graph reordering that improves locality by changing the order of processing vertices. Prior work [21, 37] has shown that traversing the edges of a graph along a Hilbert curve can improve locality of graph applications. However, these techniques complicate parallelization [5, 37]. Vertex scheduling only targets an improvement in locality and, unlike graph reordering, does not help in improving the efficiency of data duplication for power-law graphs.

Cache blocking is another technique used to improve locality of graph applications. Zhang et. al. [37] proposed CSR segmenting – a technique to improve temporal locality of `vtxData` accesses by breaking the original graph into subgraphs that fit within the Last Level Cache. Variations of cache blocking [5, 9] partition updates to `vtxData` instead of partitioning the graph. RADAR differs from these prior works in that RADAR targets not just locality improvement but also a reduction in atomic updates.

Graph partitioning techniques, traditionally used for reducing communication in distributed graph processing, have recently been applied to improve locality of in-memory graph processing frameworks [31, 32, 36]. Sun et. al. [31] proposed a partitioning approach where all the incoming edges of a vertex are placed in the same partition to improve temporal locality of memory accesses. The

authors propose modifications to the graph data structure and computation to handle a large number of partitions and demonstrate significant locality improvements along with eliminating atomic updates. RADAR aims to achieve the same goals as this prior work, but without requiring changes to the graph data structure, increasing the memory footprint, or reducing work-efficiency.

9 CONCLUSIONS

We propose RADAR, an optimization that improves the performance of graph applications by reducing atomic updates and improving cache locality. RADAR combines the mutually-beneficial optimizations of HUBDUP and Degree Sorting, offering better performance than HUBDUP and Degree Sorting applied in isolation. Finally, RADAR is an alternative technique to the push-pull optimization used for eliminating atomic updates in graph applications. We show that by avoiding a reduction in work-efficiency and maintaining a low memory footprint, RADAR is a more generally applicable optimization for single-node graph processing.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable feedback. We thank Matei Ripeanu for shepherding our paper. This work is funded in part by National Science Foundation Award XPS-1629196.

REFERENCES

[1] [n. d.]. Graph-powered Machine Learning at Google. https://ai.googleblog.com/2016/10/graph-powered-machine-learning-at-google.html. Accessed: 2019-01-23.

[2] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 22–31.

[3] V. Balaji and B. Lucia. 2018. When is Graph Reordering an Optimization? Studying the Effect of Lightweight Graph Reordering Across Applications and Input Graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 203–214. https://doi.org/10.1109/IISWC.2018.8573478

[4] Scott Beamer, Krste Asanovic, and David Patterson. 2015. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*. IEEE, 56–65.

[5] Scott Beamer, Krste Asanović, and David Patterson. 2017. Reducing pagerank communication via propagation blocking. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 820–831.

[6] Maciej Besta and Torsten Hoefler. 2015. Accelerating irregular computations with hardware transactional memory and active messages. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 161–172.

[7] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *26th International Symposium on High-Performance Parallel and Distributed Computing (HPDCâĂŹ17)*.

[8] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World wide web*. ACM, 587–596.

[9] Daniele Buono, Fabrizio Petrini, Fabio Checconi, Xing Liu, Xinyu Que, Chris Long, and Tai-Ching Tuan. 2016. Optimizing sparse matrix-vector multiplication for large-scale data analytics. In *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 37.

[10] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 1.

[11] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1.

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

[13] Neil Zhenqiang Gong and Wenchang Xu. 2014. Reciprocal versus parasocial relationships in online social networks. *Social Network Analysis and Mining* 4, 1 (2014), 184.

[14] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.. In *OSDI*, Vol. 12. 2.

[15] Konstantinos I Karantasis, Andrew Lenharth, Donald Nguyen, María J Garzarán, and Keshav Pingali. 2014. Parallelization of reordering algorithms for bandwidth and wavefront reduction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 921–932.

[16] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*. ACM, 1343–1350.

[17] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *WWW '10: Proceedings of the 19th international conference on World wide web*. ACM, New York, NY, USA, 591–600. https://doi.org/10.1145/1772690.1772751

[18] Aapo Kyrola, Guy E Blelloch, Carlos Guestrin, et al. 2012. GraphChi: Large-Scale Graph Computation on Just a PC.. In *OSDI*, Vol. 12. 31–46.

[19] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012), 716–727.

[20] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. 2017. Garaph: Efficient gpu-accelerated graph processing on a single machine with balanced replication. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 195–207.

[21] Frank McSherry, Michael Isard, and Derek Gordon Murray. 2015. Scalability! But at what COST?. In *HotOS*.

[22] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2014. Graph structure in the web—revisited: a trick of the heavy tail. In *Proceedings of the 23rd international conference on World Wide Web*. ACM, 427–432.

[23] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In *Proceedings of the 51st annual IEEE/ACM international symposium on Microarchitecture (MICRO-51)*.

[24] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* (2010).

[25] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 456–471.

[26] Roger Pearce, Maya Gokhale, and Nancy M Amato. 2014. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*. IEEE, 549–559.

[27] Larry Rudolph and Zary Segall. 1984. Dynamic Decentralized Cache Schemes for Mimd Parallel Processors. *SIGARCH Comput. Archit. News* 12, 3 (Jan. 1984), 340–347. https://doi.org/10.1145/773453.808203

[28] Nadathur Satish, Narayanan Sundaram, Md Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. 2014. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 979–990.

[29] Mahadev Satyanarayanan. 2017. The emergence of edge computing. *Computer* 50, 1 (2017), 30–39.

[30] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, Vol. 48. ACM, 135–146.

[31] Jiawen Sun, Hans Vandierendonck, and Dimitrios S Nikolopoulos. 2017. Accelerating Graph Analytics by Utilising the Memory Locality of Graph Partitioning. In *Parallel Processing (ICPP), 2017 46th International Conference on*. IEEE, 181–190.

[32] Jiawen Sun, Hans Vandierendonck, and Dimitrios S Nikolopoulos. 2017. GraphGrind: addressing load imbalance of graph partitioning. In *Proceedings of the International Conference on Supercomputing*. ACM, 16.

[33] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1214–1225.

[34] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1813–1828.

[35] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. 2018. Minnow: Lightweight Offload Engines for Worklist Management and Worklist-Directed Prefetching. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 593–607.

[36] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware Graph-structured Analytics. *SIGPLAN Not.* 50, 8 (Jan. 2015), 183–193. https://doi.org/10.1145/2858788.2688507

[37] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia. 2017. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*. 293–302. https://doi.org/10.1109/BigData.2017.8257937