Qualifying Exam Part I
Doctor of Philosophy in Computer Science

By: Reet Barik

August 11, 2020

Abstract

Expensive atomic updates and poor cache locality are primary bottlenecks when it comes to single machine shared memory graph processing applications. One popular solution is data duplication (keeping thread local copies of all vertices) but that comes with a high memory overhead. Memory-efficient versions of this strategy comes with the added run-time cost of identifying candidate vertices. The work being reviewed in this report proposes 'RADAR' which is a novel way to address both the previously mentioned problems. This report attempts to state the problem statement, critically evaluate the proposed approach, and discuss its strengths, weaknesses, and trade-offs.

1 Introduction

With applications in path-planning, social networks analysis, graph learning, data mining, semi-supervised learning, and the likes [14] [10], graphs processing algorithms are, apart from being one of the most widely used, also one of the most workload heavy as far as data intensiveness goes. With almost half the processing time comprising of cache miss latency [1], there is a need to preserve the spatial locality while storing graphs. Otherwise the irregular memory access results in expensive atomic updates. This is another bottleneck faced by parallel graph applications as seen in [5], [6], [7], and [20]. Real world graphs presents an opportunity to address these because of the sparse presence of clusters or hubs (vertices with an inordinately high degree).

With increasing main memory capacities on single machines, graph processing algorithms are now geared more towards single machine shared memory architectures. As a result, the emphasis on atomics has increased to ensure correctness of results. One solution to getting around using atomics is data duplication wherein, thread-local copies of vertices are kept and reduction is carried out across threads to arrive at the end result. This strategy is memory intense and the way to make it memory-efficient is to do data duplication selectively. Intuitively, it can be inferred that duplication needs to be done only for hub vertices since they are accessed the most. But this memory-efficiency comes at the runtime overhead cost of identifying hub vertices. Another approach to optimize graph processing algorithms is to reorder the vertices based on degree. This assigns the hub vertices contiguous IDs in the vertex array which has shown to improve spatial locality of hub accesses [3].

The work being reviewed proposes 'RADAR' which attempts to take advantage of the mutually optimizing nature of the reordering and data duplication process. Degree sorting as a preprocessing step eliminates the runtime overhead of trying to identify the hub vertices. Carrying out data duplication for only hub vertices that meet a certain threshold decreases the memory overhead significantly. The objective of this report is to take a closer look at the problem and the motivation behind 'RADAR'. This is followed by the description of the algorithm and a discussion which involves critical evaluation of the algorithm in terms of its strength, weakness, and trade-offs.

2 Background

Graph representation in memory: The most commonly used data-structure to store graphs in shared memory frameworks is the 'Compressed Sparse Row' (CSR) representation. This format is illustrated in Figure 1. The *Coordinates* array contiguously stores the neighbor of each vertex while the *Offsets* array stores each vertex's starting offset in the Coordinates array (such that vertex i's degree can be calculated by the difference in the i-th and the $i + 1$-th entry of the Offsets array). A directed graph might be stored in the form of 2 CSRs, one for the out-neighbors and the other for the in-neighbors.

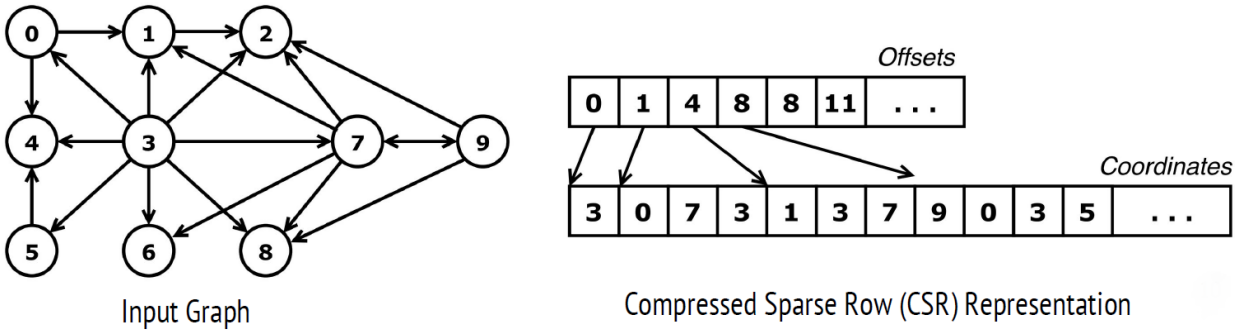


Figure 1: An illustration of the CSR representation of an example graph.

Typical Graph kernel: Graph processing algorithms are known to share a similarity as far as the core kernel is concerned. Processing of an input graph consists of visiting all vertices in the *frontier* iteratively till convergence. In shared memory frameworks, vertices are processed in parallel within iterations as shown in Algorithm 1 below:

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])
```

This is called a *push phase* execution wherein, the value of a vertex is "pushed" on to its out-neighbors. This is in contrast to the *pull phase* execution which helps to eliminate the atomic updates. As shown in Algorithm 2 below, this style of execution processes a vertex by "pulling" information from its in-neighbors. This elimination of the need for atomics comes at the cost of processing redundant edges and is hence, *work-inefficient*.

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])
```

This trade-off between the cost of atomics in push-style kernels and work-inefficiency of pull-style kernels is integrated by most graph algorithms by dynamically switching between them. For dense frontiers the pull-style kernels are preferred whereas for sparse frontiers, processing defaults to

push-style kernels (this comes with the cost of double the memory footprint since two CSRs need to be stored: one for in-neighbors and one for out-neighbors).

3 Motivation and Problem Statement

Parallel graph application frameworks in a single machine shared memory setting have been plagued by various bottlenecks. RADAR attempts to alleviate some of those. What follows makes the case for the need of something like RADAR and attempts to describe the problems it addresses:

- **Slowdown by atomics**: As shown in [20] and [6], atomic updates are a major cause of slowdown in graph applications. Figure 2 taken from [4] shows that baseline applications that replace atomics with plain loads and stores are significantly faster when compared to those with atomics.

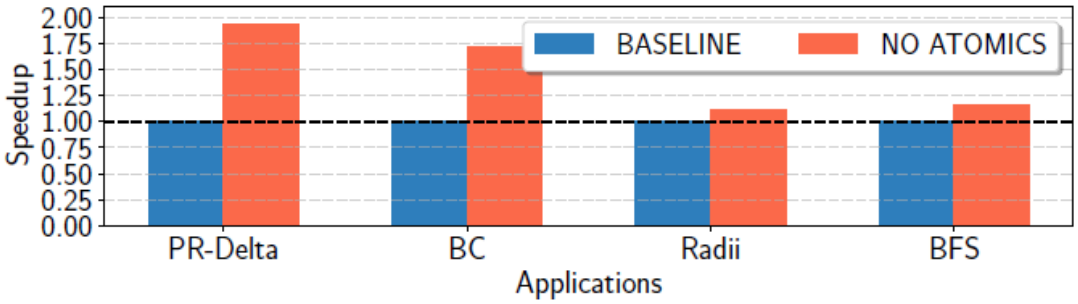


Figure 2: Net speedup obtained from replacing atomics with plain loads and stores.

- **Data Duplication as an optimization**: Data duplication has been applied as a solution for graph applications in a shared memory setting, wherein, thread-local copies of all vertex data are kept and a reduction is carried out across threads at the end to arrive at the correct result. This leads to a huge memory footprint which can be reduced by carrying out data duplication only for hub vertices (henceforth referred to as 'HUBDUP'). This reduces the memory overhead but adds the added run-time cost of having to identify hub vertices while they are being processed.

- **Locality preservation by Graph Reordering**: Graph applications suffer from irregular access to *vtxData* as shown in [5], [6], [7], and [20]. Reordering vertices by sorting them

based on degree has shown to take advantage of the power law degree distribution in real world graphs [8] and assign them contiguous IDs, thereby allowing hub vertices to fit into the cache (as shown in [3]). But as shown in Figure 3 taken from [4], a net slowdown is observed because of DegSort. This is because of *false sharing* wherein, threads comptete to share the cache lines containing the hub vertices incurring the latency of cache-coherence activity.

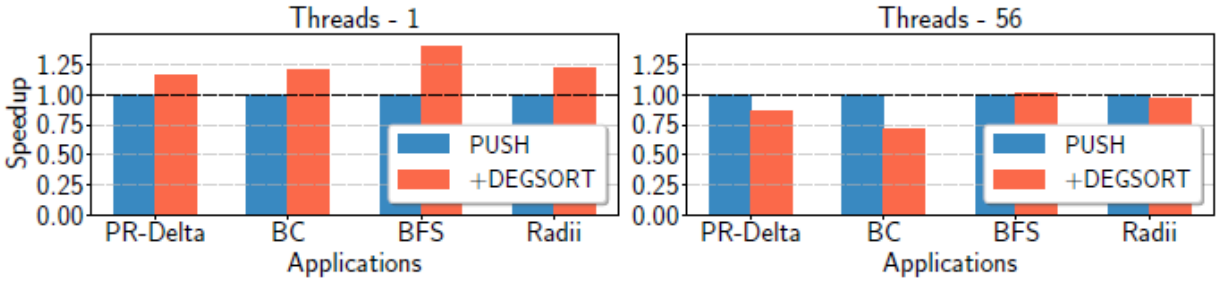


Figure 3: Net slowdown caused by DegSort in a multithreaded setting.

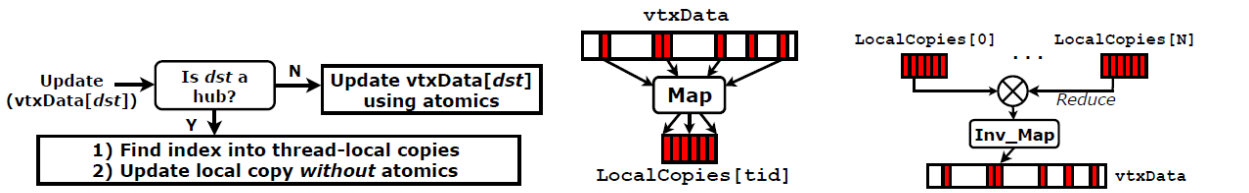
The points described above are the problems RADAR attempts to address. It does so by taking advantage of the fact that the optimizations, taken separately, though rife with their own problems are mutually enabling when made to work with each other. The following section describes how RADAR manages to do that.

4 Algorithm Description

Figure 4 describes the overall algorithm of HUBDUP. Step (a) attempts to identify if *dst* is a hub vertex on the go. If it is, thread-local copies of *dst* is made as shown in (b) and finally reduction across threads is done at the end (c) to ensure correctness of results.

With the workings of HUBDUP in mind and DegSort, the simple idea of RADAR can be summarized as follows:

$$RADAR = DegSort + HUBDUP$$



(a) Identifying hub vertices

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

(c) Reduction of values in thread-local copies

Figure 4: Conceptual description of HUBDUP

The steps for RADAR can be formulated as follows:

1. Apply DegSort on the input graph such that hub vertices get the lowest contiguous vertex IDs in the vertex array. This addresses the issue of **identifying** and **locating** the hub vertices for HUBDUP.

2. Do data duplication for HUB vertices. This is essentially the same as carrying out HUBDUP on the reordered input graph. But this time wihtout having to *map* in Step (b) or *Inv_Map* in Step (c). This is because the hub vertex IDs can be used directly as indexes into the thread-local copies (as a simple look-up).

What follows is a critique of the RADAR algorithm described above in terms of advantages, disadvantages and trade-offs.

5 Discussion

RADAR is built upon a simple idea which takes the easy identification of hub vertices by DegSort and feeds it to the memory-efficient Data Duplication scheme of HUBDUP. The advantages of the approach described in the previous section are as follows:

- **Identification of hub vertices**: Degree Sort makes it exceptionally easier to identify hub vertices by grouping them together at the beginning of the vertex array. This is a very simple way of identifying hub vertices and doesn't add to the run-time cost as was the case in naive HUBDUP.

- **Locating hub vertices**: Degree Sort assigns hub vertices contiguous IDs in the vertex array. These IDs can be directly used as indexes into the thread-local copies during hub duplication of HUBDUP without having to build a map for duplication and an inverse map during the reduction at the end.

- **Lightweight Reordering**: Degree Sort as a scheme to reorder the input graph is as lightweight a preprocessing step as they come, as opposed to heavyweight ordering schemes like [19] or [18] which would defeat the purpose of doing away with the run-time overhead of identifying and locating hub vertices.

- **Easy adaptability to varying cache sizes**: The threshold used to classify a vertex as either a hub vertex or a normal one can be used as a parameter depending on the cache size. Data duplication can be carried out only for top-k hub vertices where k is the number of vertices that can fit into a cache line (which is machine-specific).

The above advantages does come hand in hand with some disadvantages. What follows attempts to list some of the more salient ones:

- **RADAR isn't application agnostic**: Figure 5 below shows how RADAR is unequally effective for different graph algorithms. We see that it is the most effective for an application like Local Triangle Counting. This is because Local-TriCnt accesses hub vertices the most (proportionate to their degree) because hub vertices are part of most number of triangles present in the graph as opposed to normal vertices.

  As can be observed from the other two plots in Figure 5 RADAR is not as effective for applications like BFS or Radii. In fact, it is almost always as effective as standalone DegSort. This is because RADAR's locality preserving capability comes directly from DegSort. A cache line after DegSort contains hub vertices and not a single hub vertex with its neighbors. This failure to map neighborhoods in a graph to cache lines (or different levels of the cache hierarchy) by DegSort makes the *vtxData* accesses for an application like BFS, very cache-inefficient. This could be remedied by replacing DegSort with a neighborhood preserving

reordering scheme like [12], [13] or [2] which are based on community detection and graph partitioning.

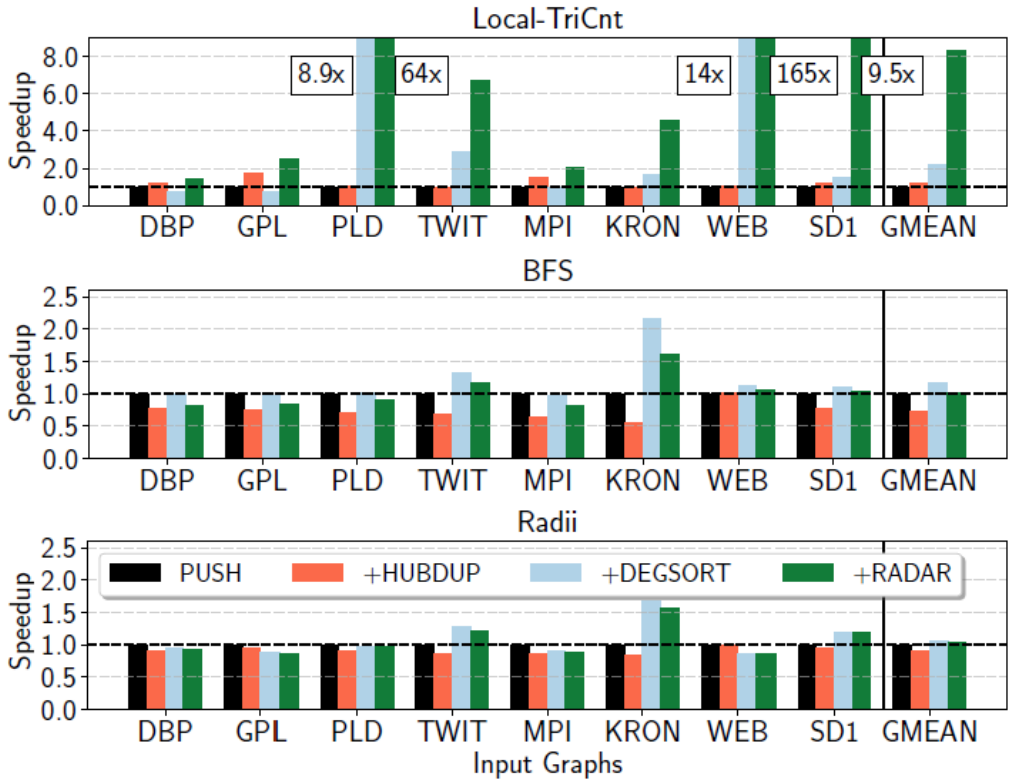


Figure 5: Varying effects of RADAR on different graph applications

- **RADAR also isn't input graph agnostic**: It can be observed from the results of [3] that lightweight reordering techniques like HubSorting and HubClustering which are minor modifications of DegSort are sensitive to the structure of the input graph. As shown in the plot in Figure 6 taken from [3], there is a direct correlation between speedup obtained from reordering techniques like DegSort and metrics like 'Packing Factor' which quantifies the graph skew and sparsity of hub vertices. It was hypothesized that input graphs meeting a certain threshold are ideal candidates for reordering with DegSort.

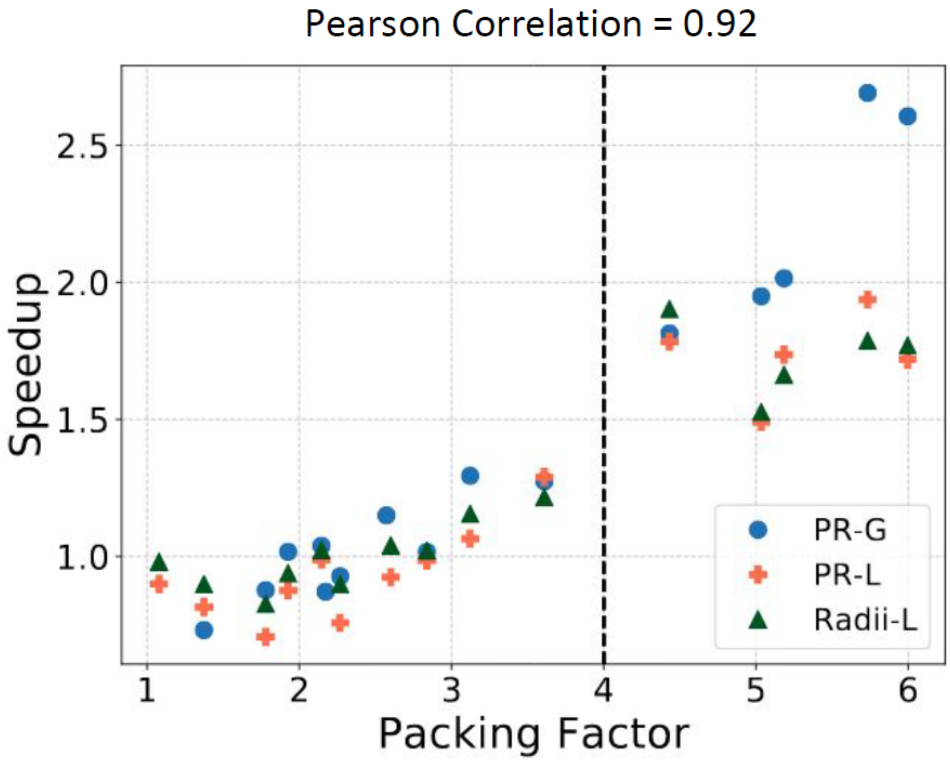


Figure 6: Correlation of Packing Factor with Speedup due to DegSort reordering.

The effectiveness of RADAR is tied very closely to the effectiveness of DegSort. Which implies RADAR performs well for those graphs which are most amenable to reordering by DegSort. This is further illustrated by the following result shown below in Figure 7.

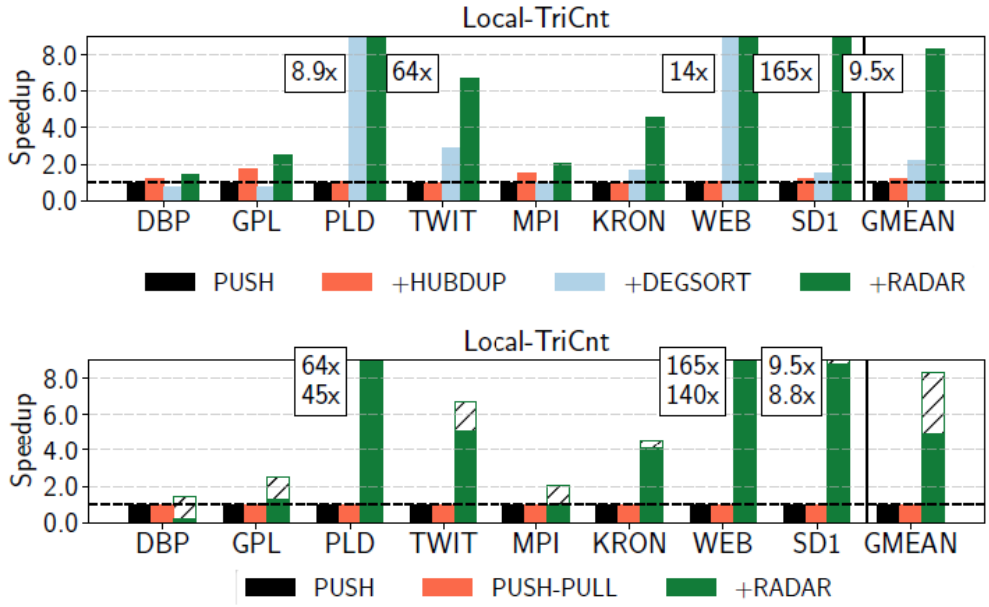


Figure 7: Effectiveness of RADAR is dependant on input graph.

For the end-application Local-TriCnt, the input graphs which show the most speedup for DegSort and RADAR as opposed to push and HUBDUP (above), are the same ones which show the most significant speedup for RADAR as opposed to Push and Push-Pull executions. This leads to the conclusion that RADAR's performance is affected by the nature of input graph.

- Another skew-aware reordering technique that is relatively as lightweight as DegSort and employs coarsegrain reordering to preserve graph structure while reducing the cache footprint of hub vertices by binning based on degree is given in [11] and could be used as an alternative to DegSort where the natural order of the input graph has been known to already preserve some form of locality.

- The end-applications that are experimented on as part of this work are traditional ones like PageRank [17], BFS [9], Triangle Counting, Radii, etc. The list of applications might be inadequate to capture the usefulness of the algorithm. Experiments on more practical end applications in the field of community detection like [15] or influence maximization like [16] might present a clearer picture of the utility of the proposed work.

6 Conclusion

This report takes a critical look at RADAR, a simple and novel idea which tries to take advantage of the mutually enabling optimizations of data duplication and graph reordering based on degree of vertices to address the bottlenecks of the same optimizations which are expensive atomics and false sharing respectively. The undeniably light-weighted-ness of RADAR, and the significant speedups observed in some of the end-applications speak to the merits of the algorithm. But a closer look suggests that the effectiveness of RADAR might be sensitive to the structure of the input graph and the access patterns of the end-applications. All in all, it provides a good starting point for the research into frameworks which try to address the issues plaguing single machine shared memory graph applications.

References

[1] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David A Wood. Dbmss on a modern processor: Where does time go? In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, number CONF, pages 266–277, 1999.

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

[3] Vignesh Balaji and Brandon Lucia. 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)*, pages 203–214. IEEE, 2018.

[4] Vignesh Balaji and Brandon Lucia. Combining data duplication and graph reordering to accelerate parallel graph processing. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 133–144, 2019.

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

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

[7] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 93–104, 2017.

[8] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009.

[9] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[10] David Easley, Jon Kleinberg, et al. Networks, crowds, and markets: Reasoning about a highly connected world. *Significance*, 9:43–44, 2012.

[11] Priyank Faldu, Jeff Diamond, and Boris Grot. A closer look at lightweight graph reordering. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–13. IEEE, 2019.

[12] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[13] George Karypis and Vipin Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[14] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600, 2010.

[15] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. Parallel heuristics for scalable community detection. *Parallel Computing*, 47:19–37, 2015.

[16] Marco Minutoli, Mahantesh Halappanavar, Ananth Kalyanaraman, Arun Sathanur, Ryan Mcclure, and Jason McDermott. Fast and scalable implementations of influence maximization algorithms. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12. IEEE, 2019.

[17] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[18] Ilya Safro, Dorit Ron, and Achi Brandt. Multilevel algorithms for linear ordering problems. *Journal of Experimental Algorithmics (JEA)*, 13:1–4, 2009.

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

[20] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching. *ACM SIGPLAN Notices*, 53(2):593–607, 2018.