Analysis of Parallel Graph Applications

Funda Atik
Computer Science and Engineering Department
Washington University in St. Louis
Doimukh, MI, USA
fundatik@gmail.com

Serif Yesil
NVidia
Urbana, IL, USA
s.serifyesil@gmail.com

Hamza Ouarnoughi
Department of Computer Science
Université Polytechnique Hauts-de-France
Valenciennes, France
Hamza.Ouarnoughi@uphf.fr

Smail Niar
LAMIH/CNRS
Université Polytechnique Hauts-de-France
Valenciennes, France
smail.niar@uphf.fr

Ozcan Ozturk
Computer Science and Engineering Program
Electronics Engineering Program
Sabanci University
Istanbul, Turkey
ozcan.ozturk@sabanciuniv.edu

***Abstract*—Despite the increasing computing power of shared memory systems with high core counts, parallel graph processing frameworks cannot exploit it effectively. The reason behind this is the inherent challenges in parallel graph algorithms, which are efficient management of dynamically created tasks and irregular data access patterns. In this paper, we categorize several popular design choices into three design dimensions: (i) execution mode, (ii) data access pattern, and (iii) work activation. We provide their high-level parallel implementations and analyze various implementations of three representative iterative graph algorithms by considering these design dimensions. To gain a better understanding of design choices, we examine their impacts on performance, communication, scalability, and work efficiency. We also investigate the communication characteristics of the design choices on two state-of-the-art shared-memory platforms by performing micro-architectural analysis. Our microarchitectural analysis reveals that a topology-driven, pull-based model gives up to 20x better performance.**

***Index Terms*—Graph Analytics, Shared Memory Systems, Parallel Frameworks, Performance, Communication.**

I. Introduction

Graphs are widely used to represent large amounts of unstructured data such as biological, road, social, and item-product networks. Many applications are designed to extract useful information from these graphs. For example, PageRank is a popular graph analytics application used to rank web pages and evaluate sentence similarity [1]. Traversal algorithms such as Breadth-First Search (BFS) and Single-Source Shortest Path (SSSP) are exercised in domains such as cognitive systems and artificial intelligence. Furthermore, collaborative filtering algorithms, namely, Stochastic Gradient Descent and Alternating Least Squares, are used in recommendation systems [2].

Designing an efficient graph algorithm executed on various graphs has many challenges due to the irregular computation patterns and data access in graph analytics applications, large graph sizes, and diversity of graph structures [3]. The underlying hardware also affects performance [4]. With the advent of big data and underlying communication requirements, these challenges are amplified, which makes designing efficient parallel graph algorithms more critical.

Shared memory implementations perform well when these graphs fit into the main memory of the system [5]–[7], whereas distributed memory settings need to amortize communication costs effectively. However, input graphs with more than a billion edges could not easily fit into the main memory of current shared memory machines. Therefore, it is necessary to understand how parallel graph applications can be implemented efficiently on shared-memory systems with large data volumes and irregular communication patterns in mind.

Due to these challenges and the popularity of graph algorithms, many parallel graph processing frameworks for shared memory systems have been proposed to lessen the programming effort when designing an efficient graph application [8]–[10]. These frameworks are powerful toolboxes that provide multiple design choices to develop graph algorithms quickly. However, designing the most performant parallel graph algorithm is still a daunting task. Each framework either implements a specific execution model or comes with a set of design choices for optimization. The best set of design decisions depends on the characteristics of the input graph, the communication requirements, the algorithm, and the underlying hardware. Often, an uninformed selection of design choices results in subpar performance. Therefore, understanding the effect of these design choices on performance and communication in graph applications is crucial.

For instance, Graphlab [8] and Pregel [10] are designed for distributed systems, but they adopt different execution models. Pregel uses synchronous execution, where data is propagated across time steps with clearly separated barriers, while Graphlab adopts asynchronous parallel execution, where the most up-to-date information is always used. From an information flow perspective, Galois [9] moves data in the outgoing direction, i.e., a push, where each node performs update operations on its outgoing neighbors. In contrast, in [11], [12], data moves in the incoming direction, i.e., pull, where each node gathers data from its incoming neighbors and performs an update operation only on itself. Furthermore, some frameworks [5], [9], [13], [14] leverage a worklist structure to drive computations via active nodes eliminating unnecessary processing for the nodes that will not contribute to the end result.

In this work, we perform a systematic analysis of these design choices on performance and communication. We implement multiple algorithms and execute on a diverse set of input graphs to test different design parameters. More specifically, we categorize the aforementioned popular design choices on shared memory systems into three orthogonal design dimensions: (i) execution mode, (ii) data activation pattern, and (iii) work activation. We discuss high-level implementations of these design decisions and implement different versions of three representative iterative graph algorithms, namely PageRank, Breadth-First Search, and Single-Source Shortest Path.

To have a better understanding of the effectiveness of different design choices, we examine their impact on performance, communication, and scalability by using both synthetic and real-world graphs. We also assess their work efficiency by considering the amount of data propagation throughout the application execution. Finally, to analyze communication characteristics, we also perform a micro-architectural analysis to investigate these design choices further. Our analysis reveals the bottlenecks of different design choices and exposes the interplay between the input graphs and the design choices.

In the rest of this paper, we first describe the basic design choices in graph frameworks in Section II.We give the graph application details in Section III. Experimental evaluation setup and results are given in Section IV and Section V, respectively. Finally, we conclude in Section VI.

II. Design Choices In Graph Frameworks

We focus on vertex-centric parallel graph processing, where a graph is the primary data structure. A graph comprises a set of vertices and edges in which edges connect vertices to represent a relationship between them. For example, in a web graph, vertices represent web pages, while edges represent the links between web pages.

In vertex-centric graph processing, applications are parallelized across vertices where the operation of a single vertex is an indivisible task. Each vertex performs a local computation by employing its value and the values of its neighbors. At the end of local computations, a vertex may update its data and/or neighbors. Such an abstraction provides easy programming. A programmer can often think of implementing these indivisible tasks as a serial program and pays very little attention to the intricacies and complexities of parallelization. Instead, a parallel graph processing framework handles parallelism for them.

A. Preliminaries

Graphs are the first-class citizens in a graph framework. A graph consists of vertices and edges where vertices connected via an edge are considered neighbors. Edges can also have directions, such as incoming edges or outgoing edges. A vertex can perform a local computation only in its neighborhood by iterating over its incoming and/or outgoing edges and neighbor vertices. Without loss of generality, we can assume that a graph framework provides a mechanism for iterating over edges and neighbor vertices of a vertex.

In this work, we focus on sparse graphs. Due to their high sparsity, they are often represented with compact data structures such as Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) representations to reduce memory footprint. We construct graphs in our implementations with CSR format because of its popularity, thanks to its minimal memory consumption. In cases where both directions are needed, we also store the transpose of the graph, which corresponds to the Compressed Sparse Column (CSC) format. Finally, a parallel graph processing framework may need to provide a worklist data structure for driving computations.

B. Design Choices

Parallel graph frameworks are an excellent toolbox for implementing graph applications without worrying about the complexities of parallel programming. Although they make programming easy by offering high-level functionalities, they either adopt different execution models or offer many execution models. However, this introduces an ample design space for optimization. A comprehensive comparison of those design choices becomes very important to make effective design choices. For this purpose, we classify several design choices that are widely used in popular frameworks in three orthogonal dimensions: (1) *execution mode*, (2) *data access pattern*, and (3) *work activation*. We explain those choices in the following.

TABLE I: Summary of design choices

	Design Choices	**Explanation**
Execution Mode	Synchronous	Updates are only visible at the next iteration, and barriers are used between iterations.
	Asynchronous	Updates are visible immediately, and atomic operations are used to enforce correctness.
Data Access Pattern	Pull-Style	Each vertex iterates over its in-neighbors and update only itself
	Push-Style	Updates itself, and can also update its out-neighbors
Work Activation	Topology-Driven	Perform operations on all vertices at every iteration
	Data-Driven	Perform operations on only a list of vertices activated in the previous iteration

The execution mode is the first design dimension, which determines the order of computations and visibility of data. We can categorize *execution modes* into *synchronous mode* and *asynchronous mode*. In *synchronous mode*, all updates computed per iteration for each vertex are only visible for computations in the next iteration, whereas, in *asynchronous mode*, all updates can be used immediately in the current iteration. *PageRank*, for example, can be implemented with *synchronous mode* by employing *Jacobi* iterations or it can be implemented with *asynchronous mode* by using *Gauss-Seidel* iterations [15].

The second design dimension is the *data access pattern* in which we have two choices: pull and push. *Pull-based* implementations iterate over incoming or outgoing edges (or neighbors) to gather data and execute a reduction operation. Note that this is a read-only operation. On the other hand, in *push-based* implementations, neighbors are updated by the vertex being processed. These write operations can be implemented with atomic operations such as a *compare-and-swap (CAS)* primitive.

The final design dimension is *work activation*, which determines whether to process all vertices at every iteration or only a subset of updated vertices. In terms of *work activation*, we can classify implementations into *topology-driven* and *data-driven*. *Topology-driven* implementations assume that all vertices in the graph are active. Thus, they process each node in every iteration without considering whether the vertices are updated. As expected, no filtering results in more computations and irregular memory accesses, causing inefficiencies. On the other hand, a *data-driven* model keeps a list of recently updated vertices, called active vertices, and only these active vertices perform local computations. This optimization typically prevents unnecessary computations and memory accesses. A summary of different design choices is presented in I.

III. Graph Applications

We implement three different graph applications: PageRank (PR), Breadth-First Search (BFS), and Single-Source Shortest Path (SSSP). Table II summarizes the design search space for three orthogonal design decisions for selected applications. We use *NA*, where specific combinations of design choices are not applicable or known to be inefficient for the application under consideration. We develop six versions of the PR algorithm by considering three design choices: the order of computations, data access patterns, and work activation. We implement four versions of SSSP and BFS by considering different combinations of data access patterns and work activations. In this section, we describe the details of these implementations.

TABLE II: Design search space for selected applications.

Work Activation	**Execution Mode**	**Access Pattern**	**PageRank**	**SSSP BFS**
Topology Driven (td)	***Synchronous (syn)***	***Pull***	tp_syn_pull	NA
		Push	NA	NA
	Asynchronous (async)	***Pull***	tp_asyn_pull	tp_pull
		Push	tp_asyn_push	tp_push
Data Driven (dd)	***Synchronous (syn)***	***Pull***	dd_syn_pull	NA
		Push	NA	NA
	Asynchronous (async)	***Pull***	dd_asyn_pull	dd_pull
		Push	dd_asyn_push	dd_push

A. Pagerank (PR)

PR is a widely adopted benchmark in many frameworks [5]–[7], [9], [12], [16] since it captures the irregular memory access, work scheduling, and load imbalance characteristics of many graph algorithms.

We first describe the *topology-driven pull-based algorithms* for *PR*. In topology-driven pull algorithms, *PR* can be performed in either *synchronous mode* or *asynchronous mode*. Equation 1 shows the calculation of a rank in a *synchronous* manner by using *Power* method [1]. In a *synchronous* implementation, it stores the current and previous vertex ranks. A vertex calculates its new rank using ranks calculated for its neighbors in the previous iteration. In this case, Pr^{t+1} and Pr^{t} are two separate data structures.

$$Pr^{t+1}[u] = \alpha \times \sum_{u \in IN(u)} \frac{Pr^{t}[u]}{T_u} + (1-\alpha) \tag{1}$$

An *asynchronous* implementation of *PR* can be realized by leveraging *Gauss-Seidel* method [15], as shown in Equation 2. For each vertex, only the most up-to-date rank is stored; thus, a vertex updates its rank by accessing the most recent ranks for its neighbors. Moreover, unlike the *synchronous mode*, there is no clear separation between iterations in *asynchronous* execution. In Equation 2, neighbors that are already processed and updated their PageRanks are shown with IN_n set, and vertices that have the Pr values from the previous iteration are shown with IN_p set. Note that, for vertex u, $IN(u)$ is the union of $IN_n(u)$ and $IN_p(u)$.

$$Pr^{t+1}[u] = \alpha \times \left(\sum_{w \in IN_n(u)} \frac{Pr^{t}[w]}{T_w} + \sum_{v \in IN_p(u)} \frac{Pr^{t+1}[v]}{T_v} \right) + (1-\alpha) \tag{2}$$

We need a worklist implementation to convert *topology-driven pull-based* algorithms to *data-driven*. For this work, we implement a worklist as a bit-vector. In addition to computations in Equations 1 and 2, each vertex checks its convergence. If the last calculation of its rank exceeds a threshold, it changes its out-neighbors' activation status by setting the corresponding bits.

The decision in which direction information flows dictates the execution mode. Synchronous and asynchronous modes can implement pull-based methods, whereas push-based methods can only be performed by asynchronous mode. Since a push-based implementation updates its outgoing neighbors' values, many vertices can try to update the same neighbor's value simultaneously when their computations overlap. We implement a push-based PR using the PR formulation described in [17]. For each vertex, we store a PR score and a residual. A vertex being processed updates uses the residual to calculate its new rank and updates its outgoing neighbors' residuals. The residual data of each vertex is transferred in each iteration instead of its score. After each node sends its residual value to its outgoing neighbors, the residual is set to zero.

B. Single-Source Shortest Path (SSSP)

SSSP aims to find a minimum cost path from a single source node to all other nodes in a weighted directed graph. In our SSSP implementations, we modify the Bellman-Ford [18] algorithm due to its adaptability to different data access patterns.

We implement four different versions of the SSSP algorithm by considering two design dimensions with different *data access pattern* and *work activation*. *SSSP* is implemented in only *asynchronous mode* since their *synchronous mode* implementations are expected to show poor performance [19], [20].

In *topology driven pull-based* implementations, shown in Equation 3, a node updates its distance by reading data (*i.e.,* pulling) from its in-neighbors.

$$dist[v] = \min\left(dist[v], \min_{u \in IN(v)} (dist[u] + weight(u \rightarrow v)\right) \quad (3)$$

On the other hand, in *push-based* variants, data flows in *OUT* direction. A node updates its outgoing neighbor's distances by transferring (*i.e.,* pushing) its distance value to its outgoing neighbors. *Push-based* applications usually generate more frequent updates. Note that updates to outgoing neighbors need to be executed atomically.

$$dist[v] = \min\left(dist[v], dist[u] + weight(u \rightarrow v)\right), \forall v \in OUT(u) \quad (4)$$

To obtain data-driven implementations, *SSSP* must be slightly modified. As in *PR*, we use a bit-vector data structure to implement a worklist. If a vertex updates its distance, it sets the bits of its outgoing neighbors. In the push version, setting outgoing neighbor's bit and distance updates can be combined.

C. Breadth-First Search (BFS)

In *Breadth-First Search (BFS)*, the goal is to find the breadth-first order traversal of the graph vertices. Similar to *SSSP*, our *BFS* implementations follow a similar logic to *Bellman-Ford* [18]. However, in *BFS*, edges in a graph do not have associated data (i.e., $weight$ in *SSSP*).

We implement all different combinations of *data access patterns* and *work activation* in *asynchronous mode*. In a *pull-based* method, a vertex executes a reduction over data of in-neighbors and finds the closest node, then updates its data accordingly. On the other hand, in a *push-based* version of *BFS*, it updates the distances of its out-neighbors atomically.

IV. Experimental Setup

A. Evaluation Platforms and Methodology

We perform experiments on two dual-socket server machines, *Haswell* and *Skylake*. Both of them have 12 cores per socket. However, memory hierarchy differs. Haswell has smaller cache sizes, 32 KB, 256KB, and 30MB caches for L1, L2, and L3 respectively. Skylake has 32KB, 1MB, and 19KB L1, L2, and L3 cache respectively. Furthermore, Skylake has non-inclusive last-level caches, while Haswell has an inclusive last-level cache.

We implement all applications in C++ and OpenMP by modifying the graph structure provided in *GAPS* [12]. We compile them using g++-4.8.5 with -O3 optimization level on CentOS Linux 7. During the experiments, we bind each thread to separate cores. If the number of threads running exceeds the number of cores in a single socket, we allocate the first half of the threads to the cores on the first socket and the remaining threads to the cores on the second socket. In the latter case, we also change the default memory interleaving policy to interleave *all* with *numactl* tool. The execution times reported are the average values of 10 runs when Turbo Boost and Hyper-threading are disabled, and each execution time only includes time for graph application and does not include time for reading the input files and constructing the graph in memory.

Input graphs are stored in the *Compressed Sparse Row (CSR)* format. For each vertex, we store their in-neighbors and out-neighbors in two different structures to improve locality. We use Likwid 4.3.4 [21] to collect performance counters for micro-architectural analysis.

B. Input Graphs

We use five large real-world graphs and generate two synthetic graphs to evaluate the impact of different design choices on the performance of the graph algorithms. Table III gives input graph details. *Kronecker (kron)* [12] and *RMAT* [22] graphs are generated synthetically with scale factor 25 and edge factor 16 by using graph converters available in *GAP* [12] and *Ligra* [5] suites, respectively. *Pay-Level Domain (pld)* [23], *First-Level Subdomain (sd1-arc)* [24], and *sk-2005 (sk)* [25] are hyperlink graphs which exhibit a power-law behavior. The first two graphs are extracted by the Common Crawl 2012 web corpus, and the third is obtained by the 2005 crawl of the *.sk* domain. We use *Twitter (tw)* graph to represent a social network. Unlike other graphs used, *USA Road (road)* [26] is a low-degree and a high-diameter graph. All input graphs are directed, and duplicate edges are removed. We use the abbreviations for the names of the graphs in all figures, which can be found in Table III.

TABLE III: Graphs used for evaluation.

Graphs	**Abbr.**	**Edges (Billions)**	**Vertices (Millions)**
USA Road	road	0.06 B	23.95 M
RMAT	rmat	0.54 B	33.55 M
Kronecker	kron	0.54 B	33.55 M
Pay-Level-Domain	pld	0.62 B	42.89 M
Twitter	tw	1.47 B	61.58 M
sk-2005	sk	1.93 B	50.64 M
First-Level Domain	sd1	1.94 B	94.95 M

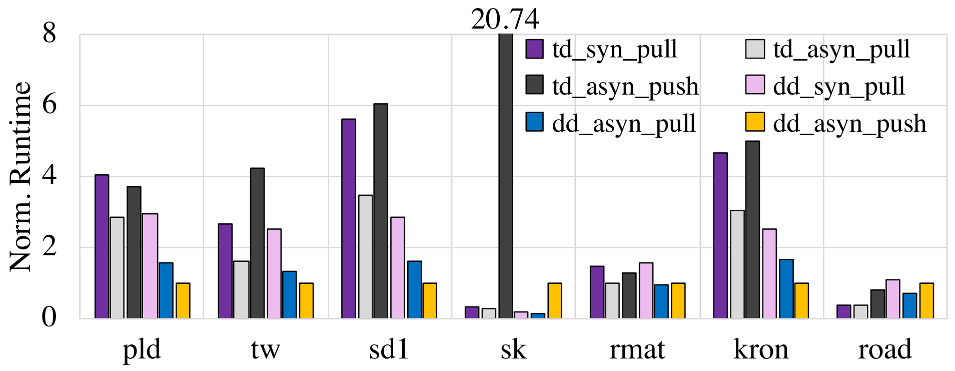
V. Experimental Results

In this section, we analyze and compare the performance of different design choices. First, we analyze runtime and scalability behavior. Secondly, we discuss the work efficiency of various implementations and their effect on performance. Finally, we provide a microarchitectural analysis of the Haswell and Skylake systems.

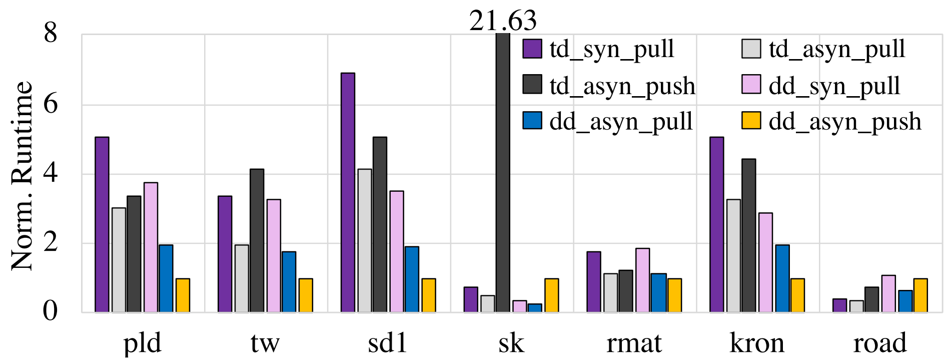
A. Runtime

Figures. 1 and 2 show the runtime of various implementations of *PR*, *BFS*, and *SSSP* by considering the different design choices on two systems.

For PR application (Figures. 1a and 1b), the data-driven implementations (dd) perform better than their topology-driven alternatives for all graphs except the *road*. In contrast, for the *road* graph, we observe that topology-driven implementation (td) is the best performer.



(a) Performance on Skylake system



(b) Performance on Haswell system

Fig. 1: Normalized execution times with 24 threads for PR application. The runtimes are normalized to dd_asyn_push implementation's execution time. Lower is better.

Specifically, we can group these graphs into three categories: (1) *pld, tw, sd1, rmat* (2) *sk*, and (3) *road*. A data-driven asynchronous push-based model is the fastest for the first group, while its pull-based counterpart gives the second-highest performance. For the *sk* graph, the two highest performances are delivered by data-driven pull-based models, whereas their push-based alternatives, especially a topology-driven one, perform very poorly. Therefore, pull-based methods (which are read-heavy) are significantly faster because read operations are much cheaper than synchronization. Furthermore, *sk* graph has a higher skew regarding the degree distribution of incoming edges.

Figure 2a and 2b show the performance of 4 different implementations of *BFS* and *SSSP* using all the cores in the system. Our experiments showed that runtime characteristics *BFS* and *SSSP* are similar on both Haswell and Skylake systems. For this reason, we only show the results for the Skylake system.

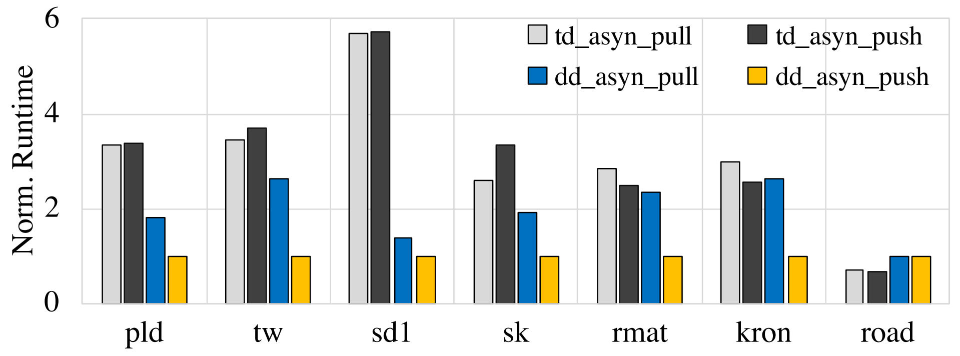
We observe that for *BFS* and *SSSP*, the *data-driven* implementations outperform *topology-driven* implementations for all graphs except the *road*. Note that in *BFS* and *SSSP*, the worklists start with a single active vertex (i.e., source vertex) and gradually increase in size up to a certain point. Therefore, topology-driven implementations perform many extra tasks. On the other hand, for the *road* graph, *data-driven* implementation performs poorly due to a lack of tasks to execute in parallel, as we will discuss shortly.

For BFS and SSSP, data propagation speed is also essential. Push-based methods for data-driven implementations outperform pull-based methods in data-driven implementation. In this case, push-based methods propagate the updates to the outgoing neighbors as soon as possible, while in the pull-based method, a vertex needs to wait until it is scheduled for execution to receive the most up-to-date value. Increasing the speed of update propagation provides significant benefits and improves convergence; thus, the overhead of synchronization is hidden.

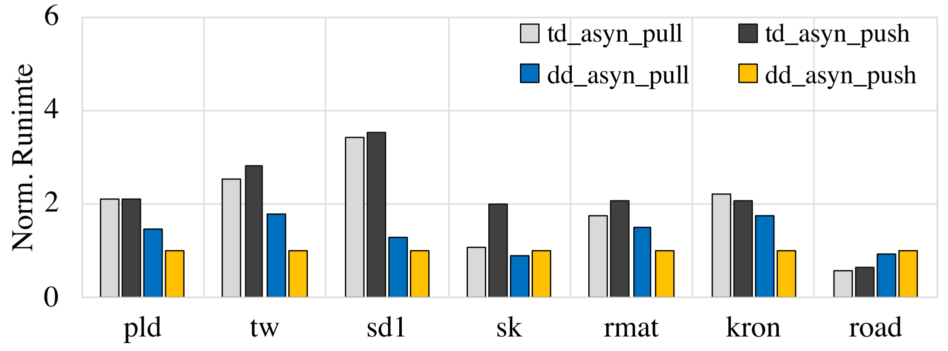
Overall, data-driven methods outperform their topology-driven alternatives. Similar to *PR*, the data-driven push-based method runs fastest, and the data-driven pull-based delivers the second-best performance for all graphs except for the *road* graph. For the *road* graph, topology-driven implementations improve performance compared to data-driven ones. In terms of the worst performance, the topology-driven push-based method gives the worst performance for *pld, tw, sd1, and sk,*, whereas its pull-based variant delivers the worst performance for *rmat* and *kron*.

B. Work Efficiency

Work efficiency is a significant factor in analyzing the impact of different design choices on the performance of algorithms. We define work efficiency as the number of useful *updates* propagated during execution. This metric can be used as a proxy for both the number of tasks that need execution and the convergence speed.



(a) Performance of BFS



(b) Performance of SSSP

Fig. 2: Normalized execution times with 24 threads for BFS and SSSP applications on Skylake system. The runtimes are normalized to dd_asyn_push implementation's execution time.

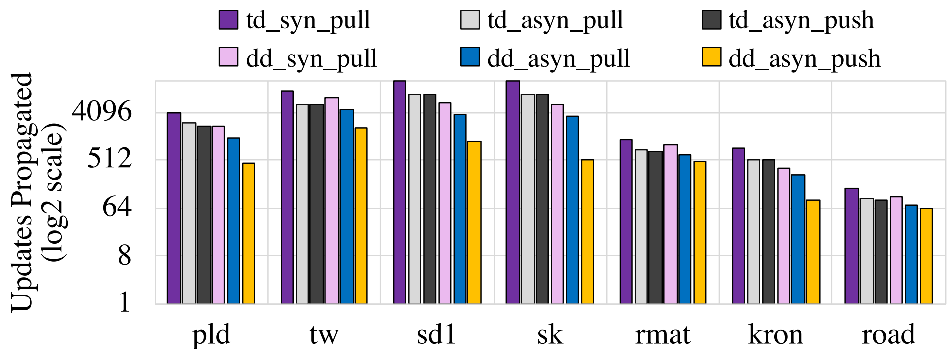


Fig. 3: Work efficiency on the Skylake machine. The total edges processed are divided by 10 million for PR variants.

For PR as shown in Figure 3, we observe significant gains from utilizing a worklist. The number of updates that need to be propagated decreases significantly when we move to data-driven implementations. The second factor in work efficiency is the speed of update propagation. Asynchronous and push-based implementations can leverage faster propagation to get better work efficiency. For PR, work efficiency pays off, as we have seen in Figures 1a and 1b, data-driven push algorithms are significantly faster.

BFS and SSSP significantly differ in their interactions with the worklist. As stated, these applications start with a single active vertex and gradually increase their worklist size. Therefore, the difference in work efficiency between a topology-driven and data-driven implementation is very significant.For example, we are switching to push-based implementations with topology-driven yields 48x better efficiency, while for data-driven, we can obtain 2.4x.

VI. Conclusions

In this work, we systematically analyzed popular design choices for parallel graph applications using three representative iterative graph algorithms, namely PageRank, Breadth-First Search, and Single-Source Shortest Path. We have analyzed their scalability, work efficiency, and communication behavior. Unlike previous work, our methodology is independent of specific benchmark suites and graph frameworks. Our analysis showed that there is no one-fits-all solution. However, it points to the importance of optimizing on-chip communication latency and throughput.

References

[1] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[2] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, Aug 2009.

[3] A. Lenharth, D. Nguyen, and K. Pingali, "Parallel graph analytics," *Commun. ACM*, vol. 59, no. 5, pp. 78–87, Apr. 2016. [Online]. Available: http://doi.acm.org/10.1145/2901919

[4] S. Beamer, "Understanding and improving graph algorithm performance," Ph.D. dissertation, EECS Department, University of California, Berkeley, Sep 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-153.html

[5] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," *SIGPLAN Not.*, vol. 48, no. 8, pp. 135–146, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2517327.2442530

[6] G. Wang, W. Xie, A. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy."

[7] D. Ediger, R. McColl, E. J. Riedy, and D. A. Bader, "STINGER: high performance data structure for streaming graphs," in *IEEE Conference on High Performance Extreme Computing, HPEC 2012, Waltham, MA, USA, September 10-12, 2012*, 2012, pp. 1–5. [Online]. Available: https://doi.org/10.1109/HPEC.2012.6408680

[8] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," *CoRR*, vol. abs/1204.6078, 2012. [Online]. Available: http://arxiv.org/abs/1204.6078

[9] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 211–222. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250759

[10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[11] A. Kusum, K. Vora, R. Gupta, and I. Neamtiu, "Efficient processing of large graphs via input reduction," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '16. New York, NY, USA: ACM, 2016, pp. 245–257. [Online]. Available: http://doi.acm.org/10.1145/2907294.2907312

[12] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[13] L. Dhulipala, G. Blelloch, and J. Shun, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '17. New York, NY, USA: ACM, 2017, pp. 293–304. [Online]. Available: http://doi.acm.org/10.1145/3087556.3087580

[14] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 12:1–12:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389013

[15] A. Arasu, J. Novak, J. Tomlin, and J. Tomlin, "Pagerank computation and the structure of the web: Experiments and algorithms," 2002.

[16] N. Sundaram, N. R. Satish, M. M. A. Patwary, S. Dulloor, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *CoRR*, vol. abs/1503.07241, 2015. [Online]. Available: http://arxiv.org/abs/1503.07241

[17] J. J. Whang, A. Lenharth, I. S. Dhillon, and K. Pingali, *Scalable Data-Driven PageRank: Algorithms, System Issues, and Lessons Learned*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 438–450. [Online]. Available: https://doi.org/10.1007/978-3-662-48096-0_34

[18] R. Bellman, "On a routmg problem," *Quart Appl Math XVI*, vol. XVI, no. no 1, 1958.

[19] D. P. Bertsekas, F. Guerriero, and R. Musmanno, "Parallel asynchronous label-correcting methods for shortest paths," *Journal of Optimization Theory and Applications*, vol. 88, no. 2, pp. 297–320, Feb 1996. [Online]. Available: https://doi.org/10.1007/BF02192173

[20] "Parallel delta-stepping algorithm for shared memory architectures," *CoRR*, vol. abs/1604.02113, 2016, withdrawn. [Online]. Available: http://arxiv.org/abs/1604.02113

[21] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

[22] "Graph500 benchmark," https://graph500.org/, accessed: 2017-12-20.

[23] "Common crawl web corpera," http://webdatacommons.org/hyperlinkgraph/, accessed: 2020-08-14.

[24] "Hyperlink graph," http://data.dws.informatik.uni-mannheim.de/hyperlinkgraph/2012-08/sd1-arc.gz, accessed: 2020-08-14.

[25] "Suitesparse matrix collection," https://sparse.tamu.edu/LAW/sk-2005, accessed: 2020-08-14.

[26] "9th dimacs implementation challenge - shortest paths," http://www.dis.uniroma1.it/challenge9/download.shtml, accessed: 2019-08-14.