Performance of Local Push Algorithms for Personalized PageRank on Multi-core Platforms

Madhav Aggarwal*§, Bingyi Zhang†§, Viktor Prasanna†
†University of Southern California, Los Angeles, USA
*National Institute of Technology, Tiruchirappalli, India
Email: aggarwal.madhav@gmail.com, bingyizh@usc.edu, prasanna@usc.edu

***Abstract*—Personalized PageRank (PPR) is used to measure the importance of vertices with respect to a source vertex. PPR is a key kernel used in many real-world applications, such as information retrieval, recommendations, knowledge discovery, etc. Local push algorithms have been widely used for developing state-of-the-art fast PPR algorithms. In this paper, we analyze the computational characteristics of local push algorithms at the algorithm (error tolerance, damping factor) and hardware architecture (available memory, cache features) levels. First, we profile the algorithm to understand the effect of various algorithm parameters. We study the trade-offs between latency and accuracy of local push algorithms using various performance metrics including Top-K accuracy and scalability. Then, we perform our analysis on two state-of-the-art multi-core platforms to understand the latency of the algorithm for PPR computation on a single source vertex and its scalability for multiple vertices using thread-level parallelism. We analyze the impact of error tolerance and damping factor on the overall performance of the algorithms.**

***Index Terms*—Personalized PageRank, local push algorithm, profiling, experimental study, multi-core platform**

I. Introduction

Personalized PageRank (PPR) is a way of modeling the importance of other vertices to a given source vertex in a graph. Using PPR, recommendation systems can identify the Top-K important vertices for a given source vertex, a process called Top-K query. Obtaining an exact solution for PPR requires matrix inversion [1], [2], which has high computation complexity. Therefore, many approximate algorithms have been proposed for fast PPR computation [3], such as Iterative Equation solving, Bookmark Coloring algorithm, etc.

Recently, local push algorithms following the message-passing paradigm have been used for developing state-of-the-art (SOTA) approximate algorithms [4]–[9]. The authors of [4]–[6] design fast algorithms that exploit the combination of local push algorithms and Monte-Carlo Sampling. Local push algorithms have unique characteristics which help in processing large graphs: (1) information propagation only happens in a small local region surrounding the source vertex (hence the name "local"), and (2) the key operations, namely graph membership queries, are required for vertex message updating.

In this paper, we conduct an experimental study to understand the characteristics of PPR computation and memory access patterns from an architectural perspective. The paper makes the following contributions. We profile the execution of local push algorithms and identify two categories of parameters that can be tuned to find highly optimized implementations of the local push algorithms: algorithm level parameters damping factor and error tolerance, and hardware architecture parameters like cache structure and size and thread count.

Our experiments show that:

- Graph membership queries are the major computation bottleneck in local push algorithms (Section III). These queries take majority ($> 75\%$) of the total execution time.
- The latency-accuracy trade-off depends on the algorithm parameters (Section IV-D). Smaller error tolerance expands the region of information propagation for processing a target vertex, leading to an increased number of graph membership queries and a higher overall latency.
- Local push algorithms have poor scalability when concurrently running PPR on multiple source or target vertices using multiple threads concurrently (Section IV-B). The poor scalability is due to the larger cache miss ratio when more threads compete for the cache (Section IV-E).

II. Background and Related Work

A. Personalized PageRank

Personalized PageRank is built upon the random walk model. Starting from the source vertex s in the input graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, at each step, the random walker has the probability α $(0 \leqslant \alpha \leqslant 1)$ of walking through any of the outgoing edges from the current position and the probability $1 - \alpha$ of stopping at the current location. Each outgoing edge has a uniform probability of being selected. Under the random walk model, $\pi(s, t)$ denotes the probability of the random walker starting from vertex s and stopping at vertex t. The notation α is called the damping factor. We use $\vec{\pi}(s)$ to denote the PageRank vector for the source vertex s and the position t of $\vec{\pi}(s)$ is $\pi(s, t)$. The following equation models the behavior of the random walker starting at source vertex s:

$$\vec{\pi}(s) = \alpha \boldsymbol{B} \vec{\pi}(s) + (1 - \alpha)\vec{1}_s, \tag{1}$$

where $\boldsymbol{B}$ is the transition matrix of the input graph, and each position $\boldsymbol{B}[i, j]$ denotes the probability of the random walker, walking from vertex i to j. $\vec{1}_s$ is a one-hot vector where

§Equal contribution

TABLE I
NOTATIONS

Notations	Descriptions	Notations	Descriptions
$\mathcal{G}(\mathcal{V},\mathcal{E})$	Input graph	$\mathcal{V}$, $\|\mathcal{V}\|$	Set of vertices , # of vertices
$\mathcal{E}$ / $\|\mathcal{E}\|$	Set of edges, # of edges	$\hat{\pi}(s.v)$	Estimation of $\pi(s.v)$
$\pi(s.v)$	PPR of vertex v with respect to s	$\pi(s)$	PPR vector for a source vertex s
$\pi(,t)$	PPR vector for a target vertex t	$\hat{\pi}(s)$	Estimation of $\pi(s)$
$\hat{\pi}(,t)$	Estimation of $\pi(,t)$	ϵ	Error tolerance
α	Damping factor	d_u	Degree of vertex u
$\hat{r}(,t)$	Residual vector for the target vertex t	nnz()	Number of non-zero elements

position s has value 1. Having the PageRank vector $\vec{\pi}(s)$, we can obtain the Top-K important vertices with respect to vertex s ranked by their PageRank values. Notations have been defined in Table I.

B. Local push algorithms

Algorithm 1 ForwardPush

Input: (s, $\mathcal{G}(\mathcal{V},\mathcal{E})$, α, ϵ)
Output: The estimated PPR vector $\hat{\pi}(s)$ for source vertex s
1: Initialize the estimate vector $\hat{\pi}(s) \leftarrow \overrightarrow{0}$
2: Initialize the residual vector $\hat{r}(s) \leftarrow \overrightarrow{0}$
3: $\hat{r}(s,s) \leftarrow 1$
4: **while** $\exists u \in \mathcal{V}$ s.t. $\frac{\hat{r}(s,u)}{d_u} > \epsilon$ **do**
5: **for** each $u \rightarrow v$ **do**
6: $\hat{r}(s,v) += (1-\alpha)\hat{r}(s,u)/d_u$
7: **end for**
8: $\hat{\pi}(s,u) += \alpha\hat{r}(s,u)$
9: $\hat{r}(s,u) = 0$
10: **end while**

Algorithm 2 ReversePush

Input: (t, $\mathcal{G}(\mathcal{V},\mathcal{E})$, α, ϵ)
Output: The estimated PPR vector $\hat{\pi}(t)$ for source vertex t
1: Initialize the estimate vector $\hat{\pi}(,t) \leftarrow \overrightarrow{0}$
2: Initialize the residual vector $\hat{r}(,t) \leftarrow \overrightarrow{0}$
3: $\hat{r}(t,t) \leftarrow 1$
4: **while** $\exists u \in \mathcal{V}$ s.t. $\hat{r}(u,t) > \epsilon$ **do**
5: **for** each $v \rightarrow u$ **do**
6: $\hat{r}(v,t) += (1-\alpha)\hat{r}(u,t)/d_v$
7: **end for**
8: $\hat{\pi}(u,t) += \alpha\hat{r}(u,t)$
9: $\hat{r}(u,t) = 0$
10: **end while**

Jeh and Widom first proposed the local push algorithm [10]. Given an input graph $\mathcal{G}(\mathcal{V},\mathcal{E})$, to estimate the PPR vector $\vec{\pi}(s)$ for the source vertex s, each vertex t maintains an estimation value $\hat{\pi}(s,t)$ and a residual value $\hat{r}(s,t)$. Based on the direction of the information push operation, we divide the local push algorithms into the ForwardPush (Algorithm 1), propagation along the outgoing edges and the ReversePush (Algorithm 2), propagation along the incoming edges. Starting with a residual value 1 assigned to the source vertex: $\hat{r}(s,s) = 1$, the local push operation iteratively pushes the residual value to the neighbors from the current vertex, and at the same time, the current vertex absorbs some residual value into its estimation $\hat{\pi}(s,t)$. ϵ is the manually specified error threshold. Andersen et. al. [11] proves that after the algorithm terminates, all the vertices $v \in \mathcal{V}$ have the estimation error $|\hat{\pi}(s,v) - \pi(s,v)| \leqslant \epsilon$. There are two major features of local push algorithms: (1) Spatial Locality: In large graphs, local push algorithms usually involve vertices in a small local range surrounding the source vertex. Therefore, the estimation array and the residual array have high sparsity. (2) Graph Membership Query: Local push algorithms search for the estimation and residual values based on the index of the vertex.

TABLE II
PLATFORMS SPECIFICATIONS

	Platform 1 (Skylake)	Platform 2 (Zen)
Processor	Intel (R) Xeon Gold 5120 CPU @ 2.20 GHz 14 Cores / 28 Threads per socket 2 Sockets	AMD Ryzen Threadripper 3990X @ 2.90 GHz 64 Cores, 128 Threads 1 Socket
Cache per socket	L1d cache: 32 KB L1i cache: 32 KB L2 cache: 32 KB L3 cache: 19.25 MB	L1d cache: 32 KB L1i cache: 32 KB L2 cache: 512 KB L3 cache: 256 MB
DDR Memory	Type: DDR4 Frequency: 2400 MHz Capacity: 128 GB Bandwidth: 89 GB/s	Type: DDR4 Frequency: 3200 MHz Capacity: 256 GB Bandwidth: 102 GB/s

III. PROFILING, IMPLEMENTATIONS, OPTIMIZATIONS

In this section, we first introduce the specifications of profiling platforms. Then, we show the initial breakdown profiling results of the local push algorithms, which motivate our detailed experimental study.

A. Platforms

We profile the performance of the local push algorithms on two SOTA multi-core platforms. The specifications of the platforms are shown in Table II. For simplicity, we denote platform 1 as *Skylake* and call platform 2 as *Zen*.

We use Vtune 2021.3 [12] to profile the execution of programs on *Skylake* and AMD uProf [13] to profile the execution of programs on *Zen*. We measure memory consumption, extent of threading and parallelism, and cache miss ratio (Section IV-A) under various algorithm parameters on the two platforms.

B. Data Structures

Graph Data Structure: In ForwardPush and ReversePush algorithms, fetching the neighbors of a vertex is the basic operation. To efficiently support the neighborhood fetching operations, we use the Adjacency list format to store the input graphs. For undirected graphs, each vertex v has a single adjacency list $\mathcal{N}(v)$ to store its neighborhood. For directed graphs, each vertex has two adjacency lists $\mathcal{N}_{in}(v)$

TABLE III
DATASET STATISTICS

Name	# of Nodes	# of Edges	Type
DBLP	613,586	1,990,159	undirected
LiveJournal	4,846,609	68,475,391	directed
Orkut	3,072,441	117,185,083	undirected
Web Stanford	281,904	2,312,497	directed
Pokec	1,632,803	30,622,564	directed

and $\mathcal{N}_{out}(v)$. $\mathcal{N}_{in}(v)$ stores the neighbors connected by the incoming edges of v and $\mathcal{N}_{out}(v)$ stores the neighbors connected by the outgoing edges of v.

C. Benchmark Datasets

We conduct experiments on five commonly used datasets [14]: DBLP [15], Web Stanford (Web-st) [16], Orkut [15], Live Journal (LJ) [17], Pokec [18]. DBLP and Orkut are undirected graphs and the other three (Lj, Pokec, Web-st) are directed graphs. For each graph, we label the vertex ids from 1 to $|\mathcal{V}|$.

Graph Membership Query: Graph membership query operations index the estimation value $\hat{\pi}(s,v)$ and residual value $\hat{r}(s,v)$ from $\vec{\pi}(s)$ and $\vec{r}(s)$ based on the vertex ID of v. Generally, local push algorithms will only touch a small proportion of total vertices $\mathcal{V}$, which means that the majority of the values in vector $\vec{\pi}(s)$ ($\vec{r}(s)$) are zeros. Therefore, we exploit the hash table to store $\vec{\pi}(s)$ ($\vec{r}(s)$). We use a SOTA open-sourced hash table implementation IMAP [19]. The vertices of the input graph are mapped to the IDs: $1, 2, 3, ..., |\mathcal{V}|$. To store $\vec{\pi}(s)$ ($\vec{r}(s)$), the IMAP allocates an array having a size proportional to the number of total vertices $O(|\mathcal{V}|)$. The position t of the array stores $\hat{r}(s,t)$ ($\vec{\pi}(s,t)$) of vertex t.

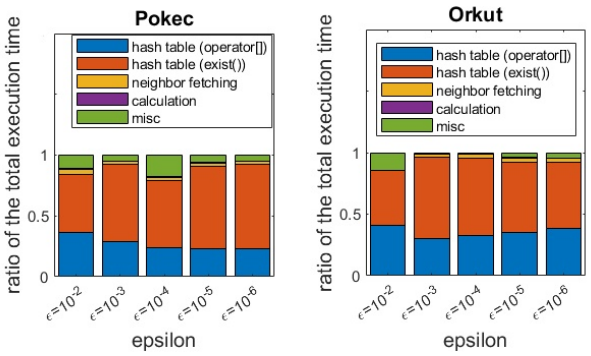

Fig. 1. Breakdown of execution time for ForwardPush algorithm on Pokec and Orkut when $\alpha = 0.2$.

To analyze the computation bottlenecks of local push algorithms, we perform initial profiling on *Skylake*. The breakdown execution time of each basic operation is shown in Figure 1. We observe that hash table operations (consist of operator [] and exist()) take more than 75% of the total execution time. Hashtable$[key]$ indexes the value of the element which has the key value key. The exist() function checks if an element is in the hash table. As demonstrated by profiling results, hash table operations are memory-bound, and $64\% - 82\%$ of the execution time is caused by memory load/store instructions of hash functions. Besides hash table operations, neighborhood fetching and calculation of residual values also consume a non-negligible portion of the total execution time. In conclusion, the profiling results show that *hash table operations are the major bottleneck in local push algorithms.*

IV. Metrics and Analysis

A. Performance Metrics

We define several performance metrics that are used in our experiments:

Throughput: The number of source or target vertices that can be processed by the platform given a specific time period. Throughput is calculated by:

$$\text{Throughput} = \frac{\text{\# of source vertices processed}}{\text{Execution Time}} \tag{2}$$

Top-K Accuracy: After computing the estimation array $\hat{\pi}(s)$ ($\hat{\pi}(,t)$), the largest top K values in $\hat{\pi}(s)$ ($\hat{\pi}(,t)$) correspond to the Top-K important vertices with respect to the source vertex s (destination vertex t). $T^{\text{K}}(s)$ ($T^{\text{K}}(t)$) denotes the set of estimated Top-K important vertices and $G^{\text{K}}(s)$ ($G^{\text{K}}(t)$) denotes the ground truth. The ground truth is calculated using matrix inversion (Equation1). The Top-K accuracy is calculated by:

$$\begin{aligned}\text{Top-K Accuracy} &= \frac{|T^{\text{K}}(s) \cap G^{\text{K}}(s)|}{|G^{\text{K}}(s)|} \times 100\% \text{ or} \\ \text{Top-K Accuracy} &= \frac{|T^{\text{K}}(t) \cap G^{\text{K}}(t)|}{|G^{\text{K}}(t)|} \times 100\%\end{aligned} \tag{3}$$

Note that when evaluating the Top-K accuracy, we set the same damping factor for $T^{\text{K}}(s)$ ($T^{\text{K}}(t)$) and $G^{\text{K}}(s)$ ($T^{\text{K}}(t)$).

Memory Usage: The main memory usage is to store non-zero elements in $\hat{\pi}(s)$ and $\hat{r}(s)$. $nnz(\hat{r}(s))$ and $nnz(\hat{\pi}(s))$ are called as the *number of vertices touched by the algorithm*. We measure the memory usage of local push algorithms using $nnz(\hat{r}(s))$ and $nnz(\hat{\pi}(s))$ after the algorithm terminates.

Cache miss ratio: It is calculated by dividing the number of cache misses by the total number of load/store operations.

B. Multi-thread Implementation

Algorithm 3 Parallel Multi-source ForwardPush

Input: p: number of threads, $\mathcal{G}(\mathcal{V},\mathcal{E})$: input graph, α, ϵ
Output: Estimated PPR vector for each vertex

```
for i = 1 : |V|/p do
    for each s ∈ V[i*p+1 : (i+1)*p] Parallel do
        π̂(s) = ForwardPush(s, G(V,E), α, ε)
    end for
end for
```

To evaluate the scalability of local push algorithms, we execute the ForwardPush and ReversePush algorithms for multiple source vertices in parallel. The parallel multi-source ForwardPush and ReversePush algorithms are shown in Algorithm 3

Algorithm 4 Parallel Multi-target ReversePush

Input: p: number of threads, $\mathcal{G}(\mathcal{V}, \mathcal{E})$: input graph, α, ϵ
Output: Estimated PPR vector for each vertex

```
1: for i = 1 : |V|/p do
2:     for each t ∈ V[i * p + 1 : (i + 1) * p] Parallel do
3:         π̂(, t) = ReversePush(t, G(V, E), α, ε)
4:     end for
5: end for
```

and Algorithm 4 respectively. We use the OpenMP library to parallelize the execution of multiple source or target vertices. We remove any potential memory initialization overheads in the evaluation results.

C. Comparison of ForwardPush and ReversePush

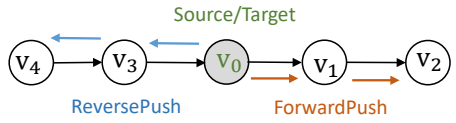

Fig. 2. ForwardPush operation and ReversePush operation in a directed graph.

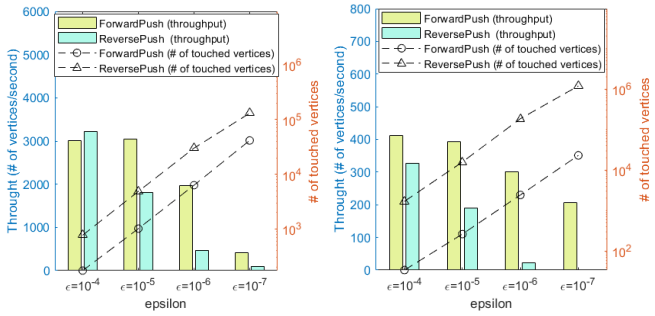

Fig. 3. Comparison of ForwardPush and ReversePush: Throughput and # of touched vertices of a) Orkut b) DBLP datasets ($\alpha = 0.2$).

ForwardPush algorithm estimates the probabilities $\{\hat{\pi}(s, v) : v \in \mathcal{V}\}$ of the random walker stopping at other vertices starting from the source vertex s. ReversePush algorithm estimates the probabilities $\{\hat{\pi}(v, t) : v \in \mathcal{V}\}$ of the random walker starting from other vertices and stopping at the target vertex. Essentially, $\hat{\pi}(s, v) \neq \hat{\pi}(v, t)$ when $s = t$, because ForwardPush and ReversePush propagate residuals along different directions as demonstrated in Algorithm 2. Therefore, it is not feasible to compare ForwardPush and ReversePush from an algorithmic perspective. Nevertheless, it is still possible to evaluate their performance using undirected graphs since the two algorithms behave similarly for these graphs.

$$\lim_{\epsilon \to 0} |\hat{\pi}(s, v) - \hat{\pi}(v, t)| = 0, \text{when } s = t.$$

We evaluate the ForwardPush and ReversePush algorithms on two undirected graphs – Orkut and DBLP. From the evaluation results (Figure 4), we observe that under the same algorithm parameters (α, ϵ), ReversePush has a higher accuracy than the ForwardPush algorithm. This is because the ReversePush algorithm has a stricter convergence criteria, as shown in line 4 of Algorithm 1 and Algorithm 2. Because of the stricter convergence criteria, the ReversePush algorithm touches more vertices and has more execution iterations. Therefore, ReversePush shows lower throughput (Figure 3) under the same algorithm parameters and platforms.

D. Algorithm Parameters

Error tolerance ϵ and damping factor α are two major algorithm parameters of local push algorithms. We evaluate how the two parameters affect the Top-K accuracy and the local push algorithms' average number of touched vertices. For simplicity, we only show the evaluation results of ForwardPush, since ReversePush follows a similar trend.

Error tolerance: Error tolerance ϵ is the criteria that controls the convergence of local push algorithms. We evaluate the Top-100 accuracy and the average number of touched vertices. As demonstrated in Figure 5, with smaller error tolerance, ForwardPush has higher accuracy and touches more vertices because it needs to propagate the residual to more neighbors in order to degrade the residual value. Generally, when the residual $\epsilon < 10^{-5}$, the Forward Push algorithm has $> 80\%$ Top-100 accuracy, and it can touch $> 10^3$ vertices. The number of touched vertices decides the memory consumption of $\hat{\pi}(s)$ and $\hat{r}(s)$. Therefore, a reasonable assumption is that when $\hat{\pi}(s)$ and $\hat{r}(s)$ can fit in the on-chip memory (e.g., cache) of the processor, the local push algorithms can directly fetch the elements from the hash table with low latency. When the size of $\hat{\pi}(s)$ and $\hat{r}(s)$ is larger than the on-chip memory, the processor needs to constantly load the elements of the hashtable from the external memory. To summarize, *a latency-accuracy tradeoff exists that necessitates the need to touch more vertices to get a higher accuracy.*

Damping factor: Damping factor α $(0 \leqslant \alpha \leqslant 1)$ is an application-specific parameter indicating the centrality of the source or target vertex. A higher α means that the random walker has a higher probability of jumping back to the source/target vertices. We evaluate the number of touched vertices for different damping factor values, since the number of touched vertices indicates the extent of memory consumption irrespective of other architectural parameters. As shown in Figure 6, when the Damping factor becomes larger, the local push algorithms touch a smaller number of vertices. This is because the decay of the residual is faster as the damping factor becomes larger. The execution time is proportional to the number of vertices touched. Thus, we observe very high execution times when the Damping Factor is low, which is in turn due to a higher number of touched vertices.

E. Scalability

In a production-scale data center, PPR is usually processed in a batched manner, which means that multiple threads process multiple source vertices in parallel. We evaluate the scalability of local push algorithms on the *Skylake* and *Zen* platforms. The number of parallel threads is set as

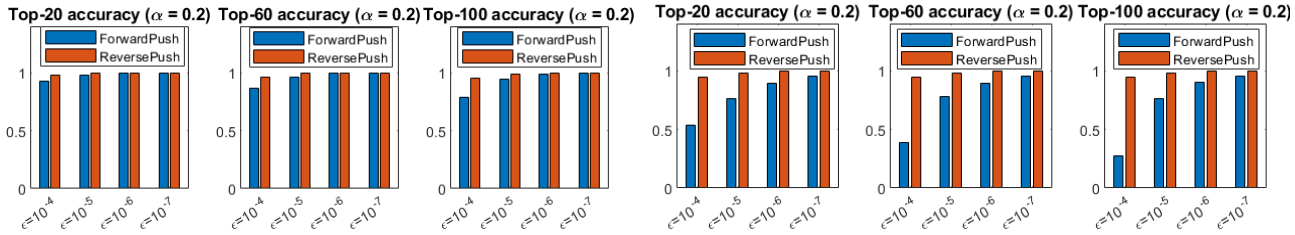

Fig. 4. Top-K ($K = 20, 60, 100$) accuracy of ForwardPush and ReversePush algorithms on Orkut (Left) and DBLP (Right) dataset.

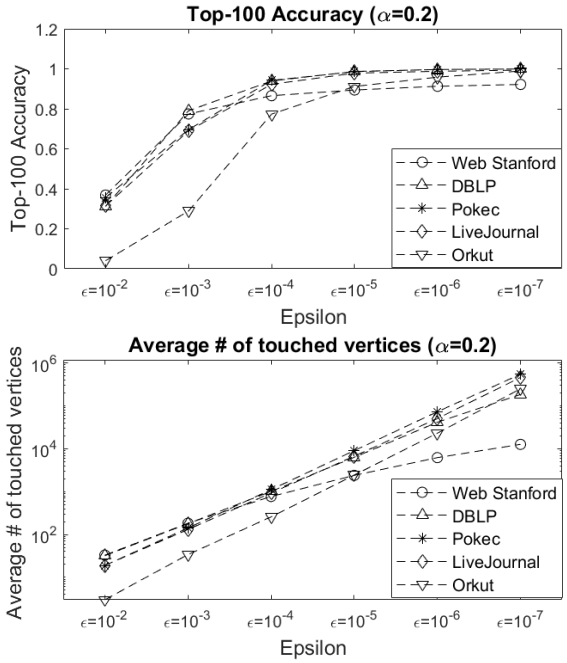

Fig. 5. Top-100 accuracy and the average # of touched vertices with respect to the error tolerance.

$1, 4, 8, 16, 32$, which is smaller than the total count of physical threads which can be used in *Skylake*. As shown in Figure 7, the throughput of PPR scales linearly when the number of threads is small. However, we observe that there is a variation in this trend when the thread count increases. We observe that when the number of threads increases from 16 to 32, there is no improvement in the throughput. The poor scalability can be explained by the cache miss ratio shown in Figure 8. As the number of parallel threads increases, the threads start to compete for the cache, leading to a significantly higher cache miss ratio.

F. Cross-platform comparison

We evaluate local push algorithms on two state-of-the-art production-scale multi-core platforms – *Skylake* and *Zen*. The comparison of throughput is shown in Figure 9. For a fair comparison, both the platforms run PPR using 32 threads (lesser than the number of physical threads). *Zen* has $0.14\times$ more memory bandwidth and a larger L2 and L3 cache size as compared to *Skylake*. Since computation takes only a small portion of the total execution time, the frequency of the computational core will not dramatically affect it. The evaluation results show that when the epsilon is within $(10^{-2}, 10^{-5})$, *Zen* achieves a speedup 1.5-$2.3\times$ that of *Skylake*. The observed speedup is partially due to the smaller cache miss ratio on the Zen platform as shown in Figure 8.

V. Conclusion and Future Directions

In this paper, we performed a detailed analysis of local push algorithms for PPR using various parameter settings. Experimental studies showed that hash table operations consume a majority of the total execution time. Based on the evaluation results, we suggest several future directions for accelerating local push algorithms:

- Efficient hash table data structures need to be developed for reducing the overhead of group membership queries.
- Some general methodologies, such as partition-centric graph processing [20] used for graph analytics can be exploited to accelerate local push algorithms.
- Apart from thread-level parallelism, fine-grained data parallelism for a single source or target vertex can be explored to reduce the execution time.

VI. Acknowledgement

We sincerely thank the anonymous reviewers for their useful comments and suggestions which helped us in improving the quality of the paper. This work was sponsored by the U.S. National Science Foundation under grant numbers OAC-1911229 and CCF-1919289.

References

[1] Y. Fujiwara, M. Nakatsuji, M. Onizuka, and M. Kitsuregawa, "Fast and exact top-k search for random walk with restart," *arXiv preprint arXiv:1201.6566*, 2012.

[2] K. Shin, J. Jung, S. Lee, and U. Kang, "Bear: Block elimination approach for random walk with restart on large graphs," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, 2015, pp. 1571–1585.

[3] S. Park, W. Lee, B. Choe, and S.-G. Lee, "A survey on personalized pagerank computation algorithms," *IEEE Access*, vol. 7, pp. 163 049–163 062, 2019.

[4] P. Lofgren, S. Banerjee, and A. Goel, "Personalized pagerank estimation and search: A bidirectional approach," in *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, 2016, pp. 163–172.

[5] S. Wang, Y. Tang, X. Xiao, Y. Yang, and Z. Li, "Hubppr: effective indexing for approximate personalized pagerank," *Proceedings of the VLDB Endowment*, vol. 10, no. 3, pp. 205–216, 2016.

[6] S. Wang, R. Yang, X. Xiao, Z. Wei, and Y. Yang, "Fora: simple and effective approximate single-source personalized pagerank," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 505–514.

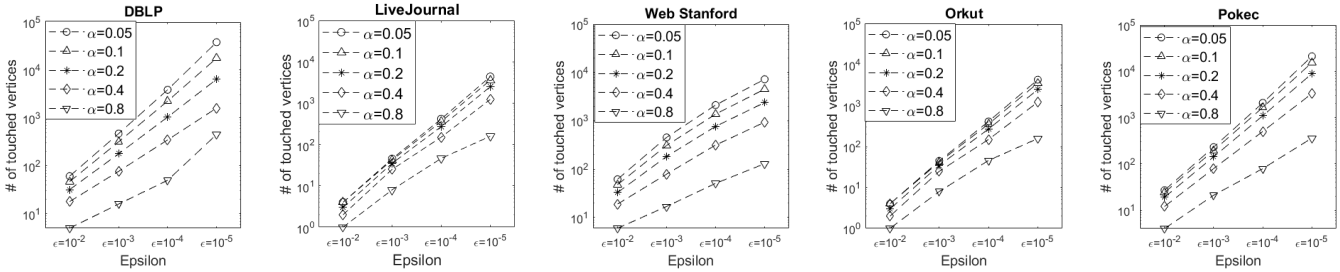

Fig. 6. The number of touched vertices with respect to dampling factor (α) for a) DBLP b) LiveJournal c) WebStanford d) Orkut e) Pokec.

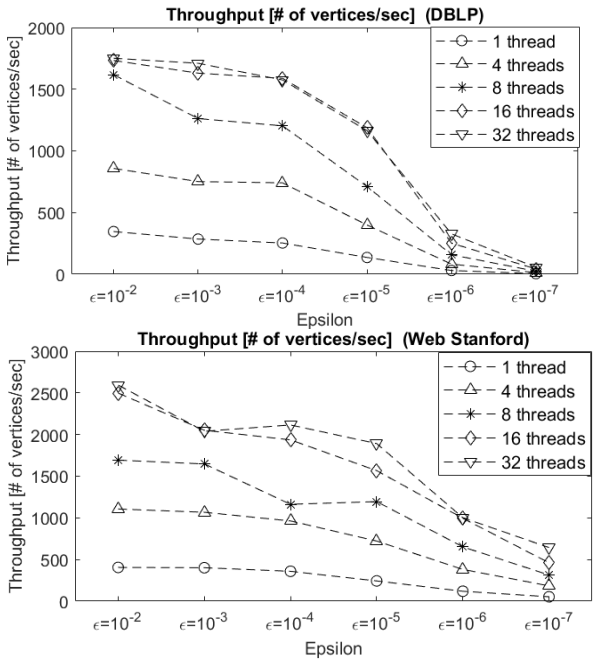

Fig. 7. Scalability of local push algorithms on *Skylake* platform.

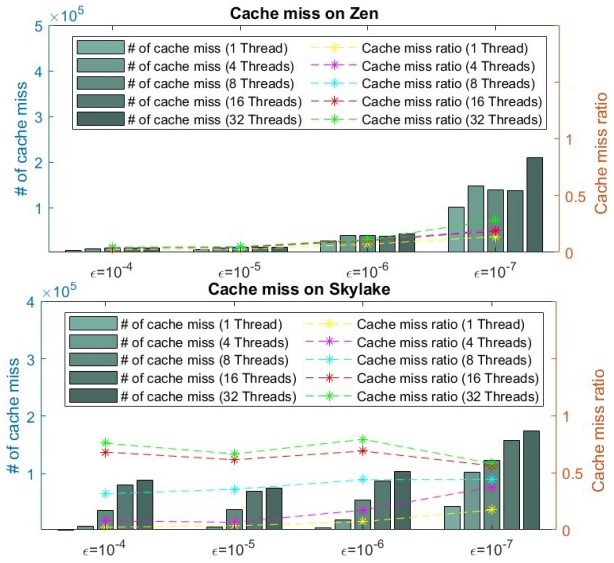

Fig. 8. Cache Miss using DBLP Dataset.

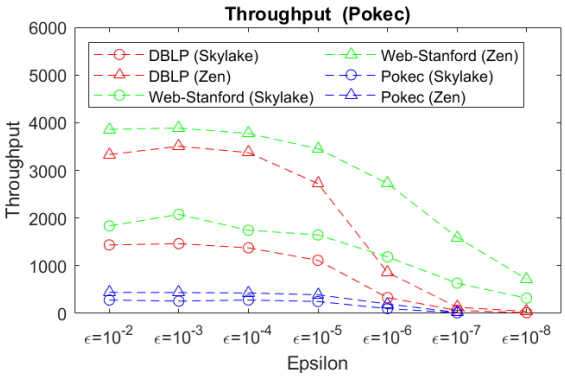

Fig. 9. Comparison of throughput on – *Skylake* and *Zen*.

[7] R. Wang, S. Wang, and X. Zhou, "Parallelizing approximate single-source personalized pagerank queries on shared memory," *The VLDB Journal*, vol. 28, no. 6, pp. 923–940, 2019.

[8] P. Lofgren, "Efficient algorithms for personalized pagerank," *arXiv preprint arXiv:1512.04633*, 2015.

[9] H. Zhang, P. Lofgren, and A. Goel, "Approximate personalized pagerank on dynamic graphs," in *Proceedings of the 22nd ACM SIGKDD International Conference on knowledge discovery and data mining*, 2016, pp. 1315–1324.

[10] G. Jeh and J. Widom, "Scaling personalized web search," in *Proceedings of the 12th international conference on World Wide Web*, 2003, pp. 271–279.

[11] R. Andersen, C. Borgs, J. Chayes, J. Hopcraft, V. S. Mirrokni, and S.-H. Teng, "Local computation of pagerank contributions," in *International Workshop on Algorithms and Models for the Web-Graph*. Springer, 2007, pp. 150–165.

[12] "Intel vtune," https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[13] "Amd uprof," https://developer.amd.com/amd-uprof/.

[14] "Stanford snap," https://snap.stanford.edu/data/.

[15] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowledge and Information Systems*, vol. 42, no. 1, pp. 181–213, 2015.

[16] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[17] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006, pp. 44–54.

[18] L. Takac and M. Zabovsky, "Data analysis in public social networks," in *International scientific conference and international workshop present day trends of innovations*, vol. 1, no. 6, 2012.

[19] "Imap," https://github.com/wangsibovictor/fora/blob/master/mylib.h.

[20] K. Lakhotia, R. Kannan, S. Pati, and V. Prasanna, "Gpop: A scalable cache-and memory-efficient framework for graph processing over parts," *ACM Transactions on Parallel Computing (TOPC)*, vol. 7, no. 1, pp. 1–24, 2020.