International Journal of Parallel Programming (2022) 50:381–404
https://doi.org/10.1007/s10766-022-00725-6

An Improved/Optimized Practical Non-Blocking PageRank Algorithm for Massive Graphs*

Hemalatha Eedi[1] · Sahith Karra[2] · Sathya Peri[1] · Neha Ranabothu[1] · Rahul Utkoor[1]

Received: 31 July 2021 / Accepted: 7 February 2022 / Published online: 26 March 2022

Abstract
PageRank kernel is a standard benchmark addressing various graph processing and analytical problems. The PageRank algorithm serves as a standard for many graph analytics and a foundation for extracting graph features and predicting user ratings in recommendation systems. The PageRank algorithm is an iterative algorithm that continuously updates the ranks of pages until it converges to a value. However, implementing the PageRank algorithm on a shared memory architecture while taking advantage of fine-grained parallelism with large-scale graphs is hard to implement. The experimental study and analysis of the parallel PageRank metric on large graphs and shared memory architectures using different programming models have been studied extensively. This paper presents the asynchronous execution of the PageRank algorithm to leverage the computations on massive graphs, especially on shared memory architectures. We evaluate the performance of our proposed non-blocking algorithms for PageRank computation on real-world and synthetic datasets using POSIX Multithreaded Library on a 56 core Intel(R) Xeon processor. We observed that our asynchronous implementations achieve $10\times$ to $30\times$ speed-up with respect to sequential runs and $5\times$ to $10\times$ improvements over synchronous variants.

Keywords PageRank · Blocking mechanism · Non-blocking mechanism · Barrier synchronization · Shared memory architecture · Multi Threading

All authors contributed equally and hence listed alphabetically.

*This work is already submitted in the 2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP).

Extended author information available on the last page of the article

1 Introduction

Graphs are widely used to represent data in various areas, including biology, genomics, astrophysics, transportation networks, web and social network analysis, and scientific computing [1]. Many of these graphs are enormous and scale to billions of nodes and edges while having uncommon and nuanced structures. As a result, numerous attempts have been made to build graph frameworks and graph libraries to solve these problems. Performance remains a significant problem in processing graphs and graph applications, especially in shared memory architectures. It is also essential to leverage the existence and interpretation of these large graphs by adding specific metrics for deriving useful analytics on many of these large graphs. PageRank is one such property that can be used to determine the quality of nodes in a web graph. Page et al. [2] devised this algorithm for Google Search Engine. The PageRank computation proceeds iteratively to estimate the significance of a web page. In each iteration, we calculate the importance of a page by randomly selecting a page and picking a random link at uniform probability d to visit another page. This process continues by updating the rank of a particular page. Pages with more links are more likely to be visited, so they eventually have higher ranks. If the outgoing link is not available, then the process moves to a new page with probability $(1 - d)$ and restarts the process from this page.

The primary understanding of the algorithm derives the rank of a page based on its incoming link. Pages that have more links are more likely to be visited so they eventually have higher ranks [2]. The rank pr of node u in Graph G is formally defined as:

$$pr(u) = \frac{1-d}{n} + d * \sum_{(v,u)\in E} \frac{pr(v)}{q} \tag{1}$$

where n is number of pages, q is outdegree defining the number of hyperlinks on page v and d is the dampening parameter initialized to 0.85.

Parallel implementations of PageRank algorithm have been extensively studied on various architectures. As PageRank algorithm iteratively progresses, multiple threads coordinate easily using synchronous mechanisms. Synchronization can be applied for both vertex-centric and edge-centric computations and on shared-memory and distributed memory architectures [3]. The barriers synchronization mechanism is more suitable for iterative algorithms such as the PageRank algorithm. However, synchronous computations utilize Thread-Level Parallelism which leads to drawbacks in dealing with progress conditions in the occurrence of thread failures. On the other hand, in asynchronous computations, progress is guaranteed where threads do not have to wait for slower threads or failure threads. This criterion motivates us to apply asynchronous computations on shared memory architecture for vertex-centric, edge-centric, or graph-centric algorithm implementations. The algorithm implementations relied on processing and computing vertices, in a Vertex-centric model [3]. Edges are the key computational units in an Edge-centric model [4]. In a Graph-centric model, the computations are performed on sub graphs with implicit compiler optimizations [5].

In this paper, we present approximation techniques for our earlier proposed non-blocking methods to leverage the computation of PageRank algorithm on massive graphs, especially on shared memory architectures. Our main focus is on designing an asynchronous PageRank algorithm with no synchronous limitations that can be applied to vertex-based and edge-based representations. We examined that applying asynchronous computations using the No-sync variant on the PageRank algorithm can speed up performance over synchronous methods. The *Loop-Perforation* is an approximate technique that skips some iterations of a loop to increase the computational speed-up. The primary idea of the loop perforation approximate technique is to reduce the amount of computation performed within each iteration as the algorithm makes progress [6, 7]. *Loop-Fusion* is an optimization technique that unites two or more independent loops into a single loop and is applied only when data dependencies are preserved. Loop fusion technique when applied increases data locality and the level of parallelism and decreases the overhead of loop control. In this direction, we applied loop perforation and loop perforation approximate technique and enabled loop fusion optimization technique to compute the PageRank algorithm.

1.1 Our Contributions

Our contributions in this work are

- Design of asynchronous techniques for iterative algorithms, especially PageRank,
- Analysis of vertex-centric and edge-centric computation on PageRank Algorithm,
- Testify the performance improvements of $5\times$ to $10\times$ speed-up when compared with synchronous variants.

2 Background and Motivation

This section gives a formal description of various synchronization approaches [8] for designing shared data objects and algorithms proposed in this paper. Also, we discuss the primary motivation to design an asynchronous technique for the iterative PageRank algorithm. In a **Shared Memory Multiprocessors or Multicore systems** multiple processors or processing elements need to coordinate accesses using shared memory. Programming implementations on shared memory systems is challenging as multiple processes simultaneously access shared resources due to lack of coordination, resulting in unpredictable delays and performance bottlenecks. An efficient synchronization mechanism is apt to deal with these issues in parallel computation by multiple processes. Two classes of synchronization approaches deal with multiple processes. (a) The **Blocking synchronization** approach uses locks to allow one thread at a time to access a shared object and thus prevents conflicts between the coordinating threads. However, it results in busy waiting and deadlocks conditions. (b) The **Non-Blocking synchronization** approach uses *lock-free* and

Wait-free methods to deal with the conflicts between the coordinating threads. When multiple threads access a shared object, the *Wait-free* approach guarantees that every thread finishes its execution in a finite number of steps. *lock-free* approach ensures that infinitely often, some thread finishes in a finite number of steps. To implement synchronization using wait-free approach, the most prominent atomic primitive used is Compare-And-Swap, CAS [8].

Algorithm 1 Bool Synch CAS(int expected, int updated)

```
1: procedure Bool Synch CAS(int expected, int updated)
2:     $temp = \frac{(1-d)}{n}$
3:     for all $v \in V$ such that $(v, u) \in E$ do
4:         $temp = temp + \frac{pr(v).load()}{outDeg(v)} * d$
5:     end for
6:     return temp
7: end procedure
```

2.1 Motivation

Most of the research done so far on PageRank computation is on the pre-processing step, i.e., processing the graph [7, 9, 10], equal distribution of load to the threads [11], etc. Most of these algorithms use a Barrier synchronization after each iteration. However Barrier synchronization has drawbacks as every thread needs to wait at each iteration and blocks indefinitely with no progress. Our main motive is to increase the computational speed by avoiding barriers and allowing the threads to run independently throughout the execution.

Designing a parallel algorithm has to solve many issues and challenges that deal with performance and memory bottlenecks. Concurrent execution by the threads to harness the underlying multi-core architecture, is an essential component that handles these challenges. In iterative algorithms, the computations of current iteration depend on the values computed from the previous iteration.

Until now, Barriers synchronization solutions achieve better parallelism on iterative algorithms. However, performance bottlenecks remain to interpret the results on scalable graphs; even though the implementation of the PageRank algorithm on shared memory architecture tends to be simpler, these solutions do not guarantee progress. Non-Blocking algorithms guarantee progress and lock-freedom/wait-freedom properties. Spotting independence across the iterations of the PageRank algorithm is non-trivial. So far, the approaches proposed for achieving better parallelism on the PageRank algorithm focuses on graph-optimization/adjacency-matrix optimization techniques. Our technique is unique, which guarantees non-blocking progress property on a PageRank algorithm. We used piece-wise concurrent programming by removing barrier constraints from an iterative algorithm and eliminating the iterations dependency.

2.2 System Model for Implementing PageRank Algorithm

We assume that our system consists of finite set of *p* threads running on multiprocessors. Threads run asynchronously and communicate with each other using shared objects. To deal with the issues raised during thread communication, we use common atomic primitives—CAS to implement wait-free algorithm and we rely on CPP vector template library to guarantee thread-safety on lock-free algorithm.

3 Related Work

This section presents an overview of the literature related to the PageRank computation from its origin to recent advances. PageRank is Google's first and previously used algorithm to rank websites in their search engine results. Page et al. [2] devised this algorithm for computing the ranks of web pages iteratively until the PageRank values converge. As it is a popular and extensively used metric to calculate the importance of web pages, there has been a lot of research interest in the past decades. Parallel computation of the PageRank metric on graphs has been studied extensively on shared memory architectures using many different programming models in recent years [1, 9, 10, 12] to mention a few.

Parallel PageRank algorithm proposed by Berry and colleagues [12] in their Multi-Threaded Graph Library (MTGL), runs on Cray XMT (Multi-Threaded Architecture extended with 128 threads) and used Q-Threads APIs for processing threads and implementing synchronization among them. Each thread computes the PR value of a node by accumulating the votes of its incoming edges of a given vertex. However, the parallel implementation of the PageRank algorithm using Q-Threads was not optimized and results in performance bottlenecks.

GraphLab—a vertex-centric programming model proposed initially on the shared-memory architecture by Low et al. [13] evolved into distributed systems for implementing parallel machine learning algorithms. The GraphLab framework was implemented in C++ using Pthreads and supports an asynchronous programming approach for computing the vertices' PageRank values by using schedulers and aggressively tuning the parameters [14]. However, in each iteration, the PageRank computations in parallel are carried out by using synchronization locks and barriers.

Wang et al., in their paper titled Asynchronous Large-Scale Graph Processing Made Easy [15], proposed Grace—a programming platform designed for shared memory systems. Grace supports synchronous iterative graph programming approach along with asynchronous features. A driver thread coordinates a group of worker threads to compute PageRank of the scheduled vertices in parallel using Barriers.

The authors of [16], discussed graph mining algorithms with a primary focus on the PageRank Algorithm. The paper aims to develop a framework for designing scalable data-driven algorithms for graph mining algorithms through a case study on the PageRank algorithm. The paper investigates various implementations of the page rank algorithm in the purview of three design axes—work activation, data access pattern, and scheduling criteria to test and understand how various design choices affect the performance. The results showed that considering data-driven designs, which are also scalable over iterative algorithms, improves performance.

The results specifically showed that the data-driven, push-back algorithmic implementations had increased the performance by 28 $\times$.

Parallel PageRank algorithm implemented using Ligra proposed by Shun and Blelloch [9] uses simple routines. It takes advantage of Frontier Based computations where an active set of vertices and edges dynamically changes through the duration of execution. To achieve parallelism, Ligra uses Clik Plus parallel codes.

Garg and Kothapalli [10] proposed four algorithmic techniques—STICD for **S**trongly connected components, **T**opological sort, **I**dentical nodes—*nodes with the same set of incoming neighbor nodes*, **C**hain nodes—*nodes with one incoming and one outgoing nod*e, and **D**ead nodes to optimize the PageRank computation by looking at the graph properties and structure. The algorithm techniques adopted in this paper exploit the nature of real-world graphs and reduces the PageRank computation by removing redundancies in edges and nodes of the graph. These kinds of optimizations can speed-up the computations when compared with the baseline parallel version. However, the preprocessing techniques used in this work are not parallelized and still need performance improvements.

The authors in the paper [17], applied an optimization technique called propagation blocking to the PageRank algorithm to reduce the memory communication bound computations, thereby improving spatial locality on DRAM. This technique is specialized to use an edge-centric representation of input data. However, the implementation is bounded by barrier synchronization.

Hamza Omar et al. in the paper [18], perform a study on the impact of input dependence for graph algorithms in the context of approximate computing. The authors justify that using perforation on the input graphs over the algorithms improves performance. Additionally, they proposed a predictor algorithm that helps in reducing the challenges in input dependencies of loop perforation for graphs and enables a satisfactory accuracy level. Experiments were tested using CPU and GPU architectures such as Nvidia, Intel CPU architectures-8 core Xeon and 61-core Xeon Phi. The results have exhibited a 30% improvement of performance on using perforation in input graphs and this, when applied to the Nvidia architecture, showed an increase of 19% of power utilization.

In [7], the authors aimed to design approximation techniques for computing, enabling good performance coupled with lesser loss of accuracy. The main techniques proposed are loop perforation, vertex/edge ordering, threshold scaling, and other heuristics such as data caching, graph coloring, etc., which are implemented and tested on the two graph algorithms, i.e., PageRank and community detection. The paper shows the performance improvement of the PageRank algorithm by 83% and up to 450$\times$ for community detection with low influence on the accuracy of using the approximation techniques on the iterative techniques. The authors conclude that approximation techniques will provide good performance with lesser loss of correctness and optimality of solutions. The performance results show 7–10 times better improvement when compared with an efficient algorithm STICD [10]. However, the approximate PageRank computation uses extra memory for storing the sorted edge-list in computing the PageRank of the target vertex. The parallel implementation still uses barriers to synchronize the computation.

The GraphPhi framework proposed by [19] mainly focuses on optimizing graph representations and uses a hybrid vertex-centric and edge-centric execution design on Intel Xeon Phi-like architectures. GraphPhi framework leverages the benefits of data-locality, effective scheduling, and load balancing. However, inspite of the advantages, the implementation is still bounded by barrier synchronization.

An optimized shared-memory graph processing framework introduce by [11] increases cache and memory efficiency. This framework is called GPOP (Graph Processing Over Partitions) framework, which promises to increase the efficiency by executing the graph algorithms at lower granularities called partitions. This framework is compared against Ligra, GraphMat, and Galois on different graph algorithms using large datasets to check the efficiency. In comparing the frameworks, GPOP shows fewer cache misses than the other frameworks and increases the performance, which is almost $19\times$ faster than Ligra, $9.3\times$- GraphMat, and $3.6\times$- Galois, respectively. The paper discusses GPOP and establishes that the framework improves cache performance, enables faster convergence, and the standard work efficiency of a given graph algorithm.

4 Description of Algorithms

This section explains the parallel PageRank computation using Blocking and Non-Blocking algorithms on large-scale graphs. Implementation of iterative parallel graph algorithms takes into consideration the following factors, like convergence, performance. We rely on convergence factor at three different levels for our algorithm: Node-level, Algorithm-level, and Thread-level. In node-level convergence, the termination of the PageRank algorithm depends on the convergence of each node independently. In algorithm-level convergence, the termination of the PageRank algorithm depends on all nodes from all partitions. In thread-level convergence, each partition terminates independently. In the below algorithms, Barriers, Barriers-Edge, Barriers-Helper fall under the algorithm-level convergence category, whereas No-Sync, No-Sync-Edge fall under thread-level convergence category. Barriers-Opt falls under a combination of node-level and algorithm-level convergence categories. No-Sync-Opt fall under a combination of node-level and thread-level convergence categories.

4.1 Barrier Algorithm

The Barriers Algorithm 2 explained here is the baseline version discussed in paper [10]. Given a graph $G = (V, E)$, vertices are divided into p equal-sized partitions. Each thread is responsible for the computation of one partition. We employed a static load allocation technique to assign nodes to partitions. Lines 4–9, to begin with, initializes all the variables and the arrays. This algorithm uses two arrays for storing PageRank values. The *prev_pr* array holds the PageRank values from the previous iteration, and the *pr* array stores the current iteration PageRank values. The error variable helps us decide if the iteration should either continue or converge. This error value is the difference between the Previous PageRank and the PageRank for each vertex. The threshold is a constant value initialized to 10^{-16}, which determines the termination condition of the algorithm.

In this algorithm the computation is divided into two phases. Lines 12–18 are the first phase of the algorithm that is responsible for PageRank computations. The algorithm computes the maximum absolute difference between the Previous PageRank and the PageRank values and saves the value in the thrErr array. Lines 20–22 are responsible for updating the shared variables. After computing the PageRank values in the current iteration, the algorithm proceeds to the next phase: to copy the values from the *pr* array to *prev_pr* array and calculate the global error value.

Algorithm 2 Barrier Baseline Algorithm

```
1: Input: $p \leftarrow \#$ of threads
2:       Graph G ← (V, E)                                   ▷ CSR representation
3: procedure ParallelPageRank($G = (V, E), p$)
4:     $error \leftarrow 1$
5:     $threshold \leftarrow 10^{-16}$
6:     for all nodes $u_i \; | i \in \{1, ..., n\}$ do
7:         $pr(u_i) \leftarrow 0$
8:         $prPrev(u_i) \leftarrow \frac{1}{n}$
9:     end for
10:    while $error > threshold$ do
11:        $thrErr[p] \leftarrow 0$                            ▷ Initialize thread's error
12:        for all $u \in threadVertices(T_i)$ do
13:            $pr(u) \leftarrow \frac{(1-d)}{n}$
14:            for all $u \in V$ such that (v,u) $\in$ E do
15:                $pr(u) = pr(u) + \frac{prPrev(v)}{outDeg(v)} * d$
16:            end for
17:            $thrErr[T_i] = max(thrErr[T_i], |prPrev(u) - pr(u)|)$
18:        end for
19:        Barrier checkpoint                                  ▷ Phase - I
20:        for all threads $T_i \; | i \in \{1, ..., p\}$ do   ▷ Update Global Error
21:            $error = max(error, thrErr[T_i])$
22:        end for
23:        $prPrev = pr$
24:        Barrier checkpoint                                  ▷ Phase - II
25:    end while
26: end procedure
```

4.2 Barrier_Edge Algorithm

Barriers-Edge is the baseline algorithm proposed in [7] paper. In this approach, the author has developed a three-phase PageRank algorithm in which the second and third phases are similar to the Barriers Algorithm. In the second phase, instead of computing the contribution values of the incoming neighbors, we directly fetch the values from the ContributionList vector. In the first phase, each node traverses through its outLinks and populates the contribution value to its respective outgoing neighbor. Similar to the Barrier Algorithm, each phase ends with barrier instruction.

Algorithm 3 Barrier-Edge Algorithm

```
1: procedure MAIN(Graph G = (V,E))
2:     G = ConvertCsr(V,E)
3:     Initialize Variables
4:     while $error > threshold$ do
5:         for all $u \in Thread\_vertices$ do
6:             if $outdeg(u) == 0$ then
7:                 continue
8:             end if
9:             $contribution = \frac{prev\_pr(u)}{outdeg(u)}$
10:            for $v \in range(outlinks[u].start, outlinks[u].end)$ do          ▷ outlinks to u
11:                $contributionList(offsetList(v)) \leftarrow contribution$
12:            end for
13:        end for
14:        barrier()                                                          ▷ Phase - I
15:        $threadError[*thdid] \leftarrow 0$
16:        for all $u \in Thread\_vertices$ do
17:            $sum \leftarrow 0$
18:            for $v \in range(inlinks[u].start, inlinks[u].end)$ do            ▷ inlinks to u
19:                $sum = sum + contributionList(v)$
20:            end for
21:            $page\_rank(u) = \frac{(1-d)}{n} + (d * sum)$
22:            $thd\_error[thd\_id] = max(thd\_error[thd\_id], prev - page_rank(u))$
23:        end for
24:        barrier()                                                          ▷ Phase - II
25:        if $thdid == 0$ then
26:            $error \leftarrow 0$
27:            for $id \in range(0, num\_of\_threads)$ do
28:                $error \leftarrow max(error, thd\_error[id])$
29:            end for
30:            $prprev = pr$
31:        end if
32:        barrier()                                                          ▷ Phase - III
33:    end while
34: end procedure
```

4.3 No_Sync

In our work, we proposed an asynchronous algorithm (No-Sync) for vertex-centric PageRank computations [20]. At least one thread should compute and update the PageRank values and make progress. Multiple threads can access the same element in this approach, but only one thread will be responsible for writing to the memory. In this process, we can encounter read–write conflicts but not write–write conflicts. A thread can read the previous PageRank value (or the one computed in the current iteration) in such a scenario. C++ vector templates guarantee this thread-safety property https://en.cppreference.com/w/cpp/container.

We modified Algorithm 2 to make it an asynchronous algorithm. The most notable change is to eliminate Barriers from the computation at the end of each phase. This change will allow threads to proceed to the next iteration without waiting for other threads to complete their task. The subsequent change reduces the memory usage by eliminating the Previous PageRank array. Since we are eliminating the iteration level dependency with our first change, we can apply our second change to Algorithm 2.

Along with the PageRank computation, each thread will compute the error value locally. In the synchronous setting, each thread will update the local error value to the global value in the second phase of computation. A thread will update its local error value and partially computed error values from other threads and enter the next iteration in an asynchronous algorithm. This property allows us to have thread-level convergence irrespective of the mode of load allocation.

Algorithm 4 No-Sync

```
1: procedure MAIN(Graph G = (V,E))
2:     G = CovertCsr(V,E)
3:     Initialize Variables
4:     while $localError > threshold$ do
5:         $thrlocErr \leftarrow 0$
6:         for all $u \in threadVertices(T_i)$ do
7:             $temp = \frac{(1-d)}{n}$
8:             $prev \leftarrow pr(u)$
9:             for all $v \in V$ such that $(v,u) \in E$ do
10:                $temp = temp + \frac{pr(v)}{outDeg(v)} * d$
11:            end for
12:            $pr(u) \leftarrow temp$
13:            $thrlocErr = max(thrlocErr, |temp - prev|)$
14:        end for
15:        $iterations[*thdid] + +$
16:        $thErr[T_i] \leftarrow thrlocErr$
17:        for all $tid \in threads(1,p)$ do
18:            $localError = max(localError, thErr[tid])$
19:        end for
20:    end while
21: end procedure
```

Lemma 1 *The algorithm eventually terminates in finite steps.*

Proof According to the algorithm, all threads will terminate when the error value of all the threads is less than the threshold. So it is enough to prove that the error value of every thread decreases in every iteration. Error value of every thread is the maximum of the error value of all the vertices that are allocated to the thread. So the problem statement boils down to prove that the error value of every vertex decreases in each iteration.

According to base algorithm, PageRank and error of vertex u in the i^{th} iteration is given by Eqs. 2 and 3 respectively.

$$pr_i^u = \frac{1-d}{n} + d * \sum_{(v,u)\in E} \frac{pr_{i-1}^v}{outDeg(v)}, \tag{2}$$

$$err_i^u = \left| pr_i^u - pr_{i-1}^u \right|. \tag{3}$$

In the No-Sync algorithm, as the threads are allowed to compute in different iterations simultaneously, at a particular instant the PageRank value of a vertex can belong to any iteration (1st to max iteration). As a base case, threads can be considered to be present in two consecutive iterations at a particular instant. Equation 2 can be modified to Eq. 4 considering that the threads are present in *ith* and $(i-1)^{th}$ iterations. Let $\mathrm{S_i^u}$ be a set of vertices where (v, u) ∈ E and PageRank of v is from *ith* iteration.

$$pr^u_{i-1:i} = \frac{1-d}{n} + d * \sum_{v \in S^u_i} \frac{pr^v_{i-1}}{outDeg(v)} + d * \sum_{v \in S^u_{i-1}} \frac{pr^v_{i-2}}{outDeg(v)}. \tag{4}$$

Error in Eq. 3 can also be modified accordingly.

$$err^u_{i-1:i} = \left|pr^u_{i-1:i} - pr^u_{i-1}\right|. \tag{5}$$

At any given instant $pr^u_{i-1} \leq pr^u_{i-1:i} \leq pr^u_i$ if $pr^u_{i-1} \leq pr^u_i$ which means $pr^u_{i-1:i}$ always lies between pr^u_i and pr^u_{i-1}.

$$\left|pr^u_{i-1:i} - pr^u_{i-1}\right| \leq \left|pr^u_i - pr^u_{i-1}\right| \Rightarrow err^u_{i-1:i} \leq err^u_i \tag{6}$$

err^u_i from the base algorithm is always expected to decrease in every iteration, so $err^u_{i-1:i}$ also decreases with every iteration. □

Lemma 2 *The algorithm leads to a similar result as that of Sequential.*

Proof PageRank of a vertex is computed from the PageRank of all its incoming vertices. As the threads are allowed to compute in different iterations simultaneously, the PageRank of a vertex can be computed from the PageRank of incoming vertices which may belong to any iteration. Equation 4 can be modified for the threads to be present in 1^{st} to i^{th} iteration.

$$\widehat{pr^u_i} = \frac{1-d}{n} + \ d * \sum_{l=1}^{I} \sum_{v \in S^u_l} \frac{\widehat{pr^v_l}}{outDeg(v)}. \tag{7}$$

The algorithm continues until the error of every node is less than the threshold, so the PageRank values of all nodes reach an almost constant value. With the given termination condition the Eq. 7 can be modified as Eq. 8 where $S^u = \bigcup_{l=1}^{I} S^u_l = \{v | (v,u) \in E\}$.

$$\widehat{pr^u} = \frac{1-d}{n} + d * \sum_{v \in S^u} \frac{\widehat{pr^v}}{outDeg(v)}. \tag{8}$$

The error obtained from the modified PageRank values is less than the threshold based on the termination condition. Hence, the PageRank values from the algorithm are also similar to that of the Sequential output with an error which is less than the threshold. Equation 8 is exactly same as Eq. 2 where $|pr^u - \widehat{pr^u}| \leq threshold$ is satisfied only at the termination condition. This lemma is also proved experimentally and the L1 norm of the PageRank values is less than 1/10th of the threshold for all the experiments. □

4.4 No_Sync_Edge

Likewise to how we developed an asynchronous algorithm for a 2-phased PageRank computational model, we also developed an asynchronous algorithm for the 3-phased PageRank computational model. The changes proposed in the previous algorithm are also applicable for this variant. In Algorithm 2, PageRank computations are happening in one single equation, whereas in Algorithm 3, Barriers-Edge, we split the equation into two parts. Though we successfully developed asynchronous variants for both algorithms, this variant does not guarantee convergence for particular types of datasets. This variant resulted in better speed-ups when we tested it on our synthetic datasets; however, it did not converge with the given threshold for standard datasets. Since the asynchronicity is entirely random, we are still exploring the reasons behind the non-convergence of this variant.

Algorithm 5 No-sync-Edge

```
1: procedure MAIN(Graph G = (V,E))
2:     G = ConvertCsr(V,E)
3:     while $error > threshold$ do
4:         $thd\_error[*thdid] \leftarrow 0$
5:         for all $u \in Thread_vertices$ do
6:             $prev \leftarrow page_rank(u)$
7:             $sum \leftarrow 0$
8:             for $v \in range(inlinks[u].start, inlinks[u].end)$ do        ▷ inlinks to u
9:                 $sum = sum + contributionList(v)$
10:            end for
11:            $page\_rank(u) = \frac{(1-d)}{n} + (d * sum)$
12:            $thd\_error[thd\_id] = max(thd\_error[thd\_id], prev - page\_rank(u))$
13:        end for
14:        $interation[*thdid] + +$
15:        $error \leftarrow 0$
16:        for ( do$id \in range(0, num\_of\_threads)$)
17:            $error \leftarrow max(error, thd\_error[id])$
18:        end for
19:        for all ( do$u \in Thread\_vertices$)
20:            if $outdeg(u) == 0$ then
21:                continue
22:            end if
23:            $contribution = \frac{page\_rank(u)}{outdeg(u)}$
24:            for $v \in range(outlinks[u].start, outlinks[u].end)$ do      ▷ outlinks to u
25:                $contributionList(offsetList(v)) \leftarrow contribution$
26:            end for
27:        end for
28:    end while
29: end procedure
```

4.5 Barrier and No_Sync Variants Optimization

Many applications might not require the exact solution which can help reduce the overall computational cost. When using approximation techniques, we skip some portions of the computation to arrive at an approximate solution. This technique can significantly improve the performance by a minimum compromise on accuracy.

Algorithm 6 Barrier Optimization and NoSync Optimization Algorithm

```
⋮
1:
2: while $error >$ $threshold$ do
3:     $thrErr[p] \leftarrow 0$                                  ▷ Initialize thread's error
4:     for all $u \in threadVertices(T_i)$ do
5:         $pr(u) \leftarrow \frac{(1-d)}{n}$
6:         if $threshold\_check[i] == False$ then
7:             for all $u \in V$ such that (v,u) ∈ E do
8:                 $pr(u) = pr(u) + contribution(v) * d$
9:             end for
10:            $thrErr[T_i] = max(thrErr[T_i], |prPrev(u) - pr(u)|)$
11:            if $|prPrev(u) - pr(u)| != 0 \&\& |prPrev(u) - pr(u)| < threshold * 0.00001$ then
12:                $threshold\_check[i] = True$
13:            end if
14:        end if
15:    end for
16: end while
⋮
```

Loop perforation is an optimization technique that can reduce iterations without changing the definite description of an algorithm when applied to iterative algorithms. We used this technique for our Barriers and No-Sync variants as proposed in paper [7] to compute the PageRank algorithm. We made a slight modification to the author's technique, where we are eliminating the PageRank computations if the absolute difference of PageRank and Previous PageRank of a node is less than 10^{-21}.

4.6 Barrier_Helper

In this Wait-free algorithm, we address thread delay/failure scenarios by ensuring the algorithm's correctness. Here the threads are not allowed to enter into the next iteration until it computes the PageRank for all nodes in that particular iteration. Any thread that finishes the computation of its allocation will help any other random thread before proceeding into the next iteration. The threads continue to help other threads in progress until PageRank of all nodes gets computed. In Algorithm 7, all available threads execute the ThreadPageRank () procedure in line 47. Each thread computes the PageRank of nodes in its partition by calling ComputePR() in line 51. After finishing its partition, threads are allowed to help incomplete threads from lines 53 and 54. Updating global variables like iteration number, error, and PageRank of Sink nodes happens in 56 to 59. Each thread has a global atomic variable (glbThdInfo), which stores the information like iteration number, latest computed node, thread error, and thread PageRank of sink nodes. This thread variable is global and accessible by every other thread. *Thd1* helping *Thd2* update the information in the *Thd2* global variable. This update of *glbThdInfo* is done from lines 24 to 28 in the ComputePR( ) procedure. UpdatePR( ) method from lines 1 to 12 is used to update the PageRank value and the iteration number using CAS operation. Every variable is associated with an iteration number to avoid any wrong updates by a slow thread present in previous iterations, where some helper thread would have already updated.

```
struct ThCASOb{
    int itr;
    int currNode;
    double thErr;
};
struct GlbCASOb{
    int itr;
    double err;
    vector<bool>check;
    bool intermediate;
};
struct PrCasOb{
    int itr;
    double rank;
};
```

5 Experiments Evaluation

5.1 Platform

We conducted our simulations on a 56 Intel(R) Xeon(R) E5-2660 v4 processor architecture with two CPU sockets. Each socket has 14 cores and two logical threads per core running at 2.06 GHz core frequency. Every core’s L1: 32K, L2: 256K cache memory is private to that core, and L3: 35840K cache memory is shared across the cores.

Algorithm 7 Wait-Free

```
procedure UPDATEPR(u, nodePr, thdVar)
    $z \leftarrow pr(u)$
    if $z.itr == thdVar.itr$ then
        $casOb \leftarrow newPrCasOb(thVar.itr++, nodePr)$
        $CAS(pr(u), z, casOb)$
    end if
    $z \leftarrow prevPr(u)$
    if $z.itr == thdVar.itr$ then
        $casOb \leftarrow newPrCasOb(thVar.itr++, nodePr)$
        $CAS(prevPr[u], z, casOb)$
    end if
end procedure
procedure COMPUTEPR(thdId, hlpId, thdVar)
    $thInfo \leftarrow glbThInfo[hlpId].load()$
    while $u \in ThreadVertices$ and $glbVar.itr == thdVar.itr$ do
        $nodePr \leftarrow \frac{(1-d)}{n}$
        for all $v \in V$ such that $(v, u) \in E$ do
            $nodePr += \dfrac{glbPrevPr[v]}{outDeg(v)} * d$
        end for
        Invoke updatePr(u,nodePr,thdVar)
        $z = glbThInfo[hlpId]$
        if $z.itr == thdVar.itr$ then
            $er \leftarrow max(z.er, |nodePr - prevPr|)$
            $casOb \leftarrow newThCASOb(z.itr, next(u, hepId), er)$
            $CAS(glbThInfo[hlpId], z, casOb)$
        end if
    end while
end procedure
procedure UPDATEGLBVAR((thId,hlpId,thdVar)
    while $true$ do
        $z \leftarrow glbVar$
        if $z.itr == thdVar.itr$ then
            $casOb \leftarrow copy(z)$
            $casOb.check[helpId] \leftarrow true$
            $casOb.er \leftarrow max(casOb.er, glbThInfo[hlpId].er)$
            $CAS(glbVar, z, casOb) break$
        end if
    end while
    while $true$ do
        $z \leftarrow glbThInfo[hlpId]$
        if $z.itr == thdVar.itr$ then
            $casOb \leftarrow newThCASOb(z.itr++, thdId, 0)$
            $CAS(glbThInfo[hlpId], z, casOb) break$
        end if
    end while
end procedure
procedure THREADPAGERANK(thdId)
    while $glbVar.load().er > threshold$ do
        Invoke ComputePR(thdId,thdId,thdVar)
        for all $thr \in threads$ and $thr != thdId$ and notCompletePR(thr) do
            Invoke ComputePR(thr,thdId,thdVar)
        end for
        Intialize error value to 0 for next iteration in glbVar using CAS
        Invoke UpdateGlbVar(thdId,thdId,thdVar)
        for all $thr \in threads$ and $thr != thdId$ and notCompleteGlbVar(thr) do
            Invoke UpdateGlbVar(thr,thdId,thdVar)
        end for
        Intialize itr in glbVar using CAS
        $thdVar \leftarrow glbVar.load()$
        Invoke Swap()
    end while
end procedure
```

All the simulations were coded in C/C++, compiled using g++ 7.5.0 and the POSIX Multi-Threaded library. We also compared our experiments using OpenMP APIs.

5.2 Datasets

We use both synthetic datasets and four categories of real-world datasets in our simulations, as listed in Table 1. The datasets are chosen, ensuring the related studies [9, 10, 21] in providing a fair comparison. We conduct initial experiments on randomly generated synthetic graphs in the range of $1 \times 10^6 to 7 \times 10^6$ edges using RMAT graph library [22]. Later on, Web-Graphs, Social-Networks, and Road-Networks, from standard datasets repository [23]. All the [21] graph datasets sizes are in Adjacency List format, which is later converted to both CSR (Compressed Sparse Row) format and an edge representation. We tested all the proposed algorithms on the given datasets.

5.3 Results and Discussion

In this section, we present the speed-up achieved by parallel variants of PageRank algorithms. The ratio between the Sequential execution time (the original PageRank algorithm)and Parallel execution time is the metric for calculating the algorithm's speedup. With a fixed number of threads (56) on a different class of datasets, we

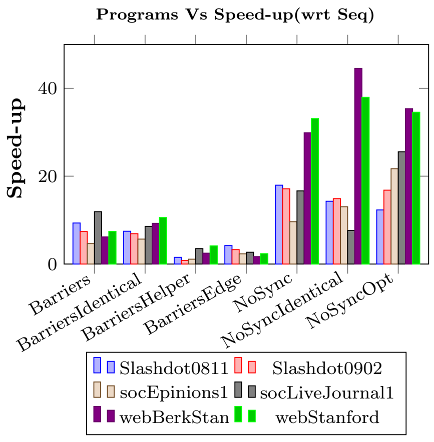
Table 1 Real-world and synthetic graph datasets

Datasets	#Vertices	#Edges	Size (MB)
Web Graphs [23]			
web-Stanford	281,903	2,312,497	30
web-NotreDame	325,729	1,497,134	20
web-BerkStan	685,230	7,600,595	20
web-Google	875,713	5,105,039	7
Social Networks [24]			
soc-Epinions1	75,879	508,837	5.7
Slashdot0811	77,360	905,468	10.7
Slashdot0902	82,168	948,464	11.3
soc-LiveJournal1	4,847,571	68,993,773	1100
Road Networks [24]			
road-italy-osm	6,686,493	7,013,978	109.9
great-britain-osm	7.7M	8.2M	28
asia-osm	12M	12.7M	5.1
germany-osm	11.5M	12.4M	98.5
Synthetic Graphs [21, 22]			
D10	491,550	0,999,999	13.2
D20	954,225	1,999,999	28.3
D30	1,400,539	2,999,999	43.3
D40	1,871,477	3,999,999	59.0
D50	2,303,074	4,999,999	74.1
D60	2,759,417	5,999,999	89.9
D70	3,222,209	6,999,999	105.6

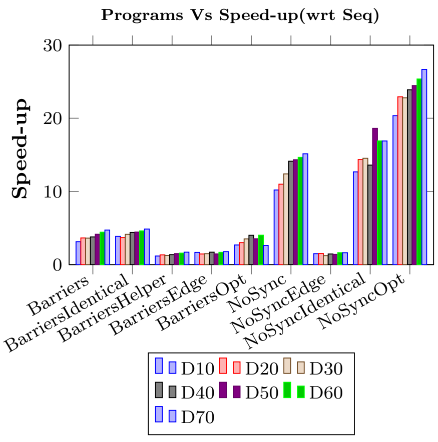
execute the programs and obtain the execution times. When incorporated with existing graph processing methods, the proposed algorithms prove significant improvement at the hardware level.

Figure 1a shows the speed-ups obtained by parallel variants (blocking and non-blocking variants) on standard datasets using 56 threads. Barrier variants result in a maximum of 10× on standard datasets, whereas No-Sync variants (except for No-Sync-Edge) consistently produce greater than 10× speed-up on almost all datasets. It is observed from the results that No-Sync, No-Sync-Identical, No-Sync-Opt and No-Sync-Opt-Identical, are performing better than the Barriers, Barriers-Identical and Barriers-Edge on all the datasets. We achieve substantial performance benefits by removing the barriers and allowing partial computations on shared variables to eliminate iteration-level dependency and thread-level dependency. Thus it can be concluded that asynchronous variants outperform synchronous variants by a considerable magnitude. As each thread progresses independently and completes the given task, we achieve the lock-free property on the No-Sync variants. We conclude that the lock-free variants of the PageRank algorithm provide better performance improvements compared to the other variants. The notion behind the Wait-free variant is to display the sustainability of the current program execution and hence is not explicitly designed for performance. Since we are not using any compiler optimization flags, the Barriers-Edge variant is not as performant. Figure 1b shows the speed-ups obtained by parallel variants on synthetic datasets. The insights noted in Fig. 1a are also applicable here for synthetic datasets. Barrier variants result in a maximum of 5× speed-up on synthetic datasets, whereas No-Sync variants (except for No-Sync-Edge) consistently produce greater than 10× speed-up on almost all datasets. It is observed for Synthetic datasets that as size increases, No-Sync variants consistently outperform Barriers variants in terms of performance.

Figure 2a, b show the speed-ups obtained by parallel variants by varying the threads on randomly selected datasets (web-Stanford a standard dataset and D70 a



(a) Standard Datasets



(b) Synthetic Datasets

Fig. 1 Speed-Up vs. Programs on Standard and Synthetic Datasets (56 Threads)

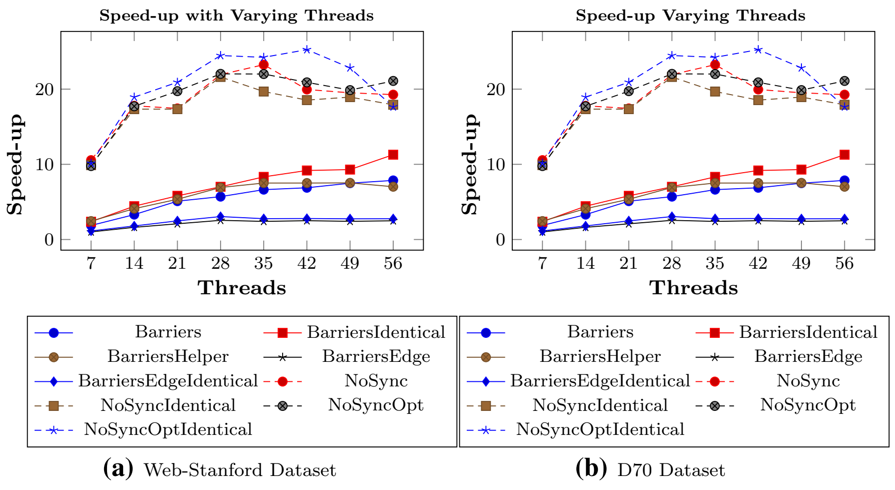


Fig. 2 PageRank Speed-Up with Varying threads for web-Stanford Dataset updated

synthetic dataset). In this work, we apply the static load balancing technique to all parallel variants. With an increase in the number of threads, the speed-up rate is significantly less for barrier variants than the No-Sync variants since each thread has to wait for others in the barrier variants. This also leads No-Sync variants to have much better scalability in comparison to barrier variants. On the other hand, in No-Sync variants, as each thread progresses independently, we achieve a higher speed-up with a higher thread value. These results suggest, our lock-free variant scales well with the increase in the number of threads.

Figure 3a, b show the speed-up and L1-norm obtained by parallel variants on a randomly selected dataset (web-Stanford a standard dataset and D70 a synthetic

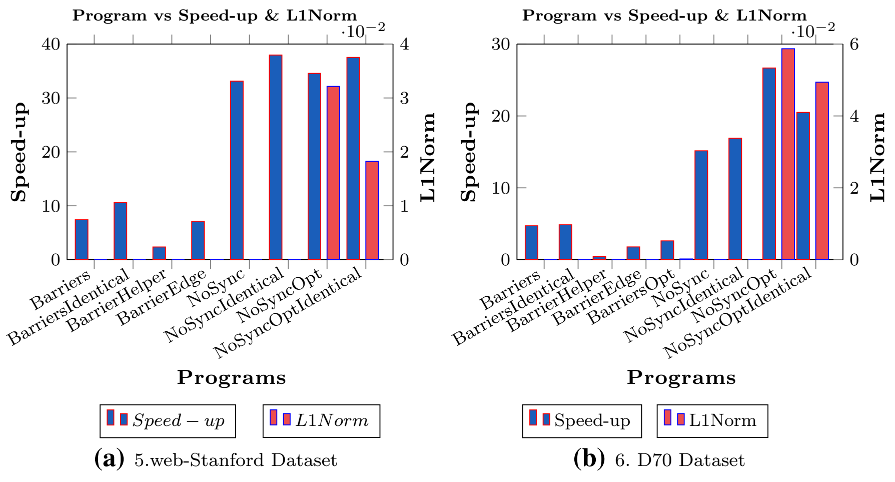


Fig. 3 PageRank Speed-Up and L1-norm)

dataset) with a fixed thread count (fixed at 56). The summation of differences between PageRank of each node from sequential and parallel variants denotes L1-norm. For most Barrier variants, the L1-norm is zero, which means the page rank values are equal to the sequential ones. No-Sync algorithms, except approximation algorithms on all datasets, is achieving a zero L1-norm. The value is high for No-Sync-Opt and No-Sync-Opt-Identical as we are performing the loop-perforation technique and skipping the computations when its PageRank value is less than 10^{-21}. The result of using the above approximation technique leads to an increase in speed-up and L1-norm.

In Fig. 4, we compare the number of iterations taken by each parallel variant. Ideally, we expect each variant to achieve convergence with the same number of iterations. In our case, as we are allowing threads to do partial updates on shared variables that depend on the convergence, No-Sync variants are taking a fewer number of iterations than barrier variants. Our lock-free variant not only gives better speed-up but it also converges faster. Prior to this work, we knew about node-level convergence and algorithm-level convergence on the iterative algorithm, but to our knowledge, we are the first ones to propose thread-level convergence.

Sleeping variants to evaluate the impact of the Wait-free algorithm, we deterministically added sleep to the threads in selected iterations. In the Barrier algorithm, each thread has to wait until the completion of the sleeping thread. In No-Sync, the work corresponding to the sleeping thread will be resumed after the thread awakes. Our Wait-free (Barrier-helper) algorithm is robust enough to handle the above two drawbacks. In the case of a Wait-free algorithm, a thread will not wait for another thread and helps other threads after completing the assigned task. In Fig. 5, we can see the execution times of Barriers and No-Sync algorithms are increasing with an increase in sleep time, whereas Wait-free execution time is consistent.

Failing variants except for Wait-free, other parallel variants do not handle thread failures. We deterministically added failures to the threads after the end of the first iteration to evaluate its impact. In Fig. 6, we can see the increase in the program execution time as we increase thread failures.

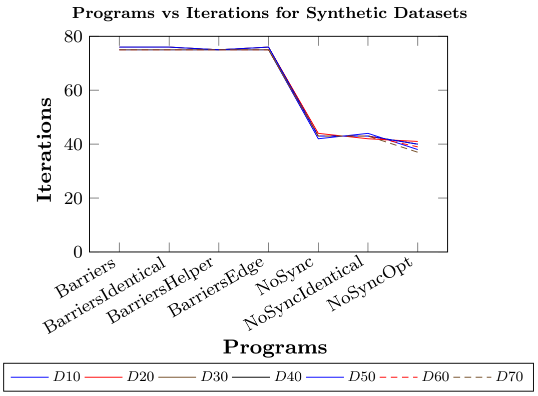


Fig. 4 Program vs. # of Iterations on Synthetic Datasets (56 Threads)

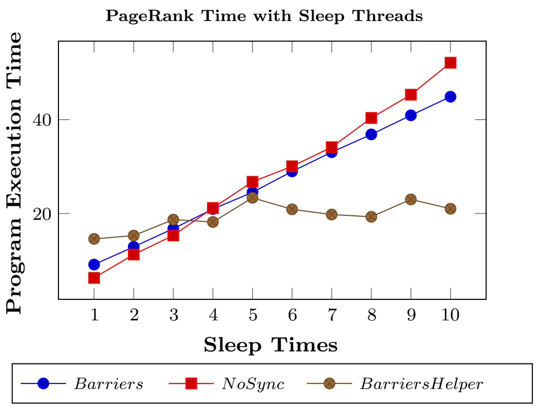


Fig. 5 Simulation results for the sleeper threads

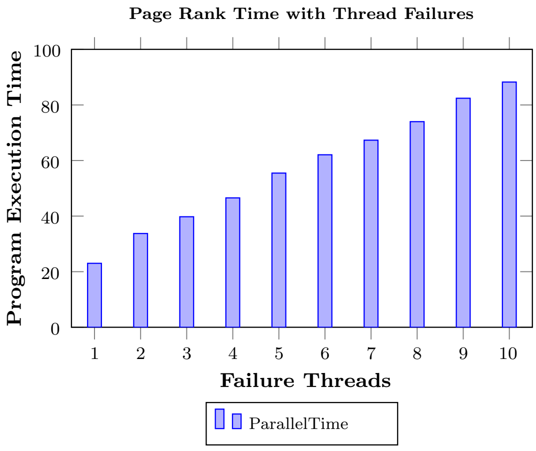


Fig. 6 Simulation results for the failure threads

5.4 Comparative Results using OpenMP

OpenMP and POSIX Pthreads are two different paradigms for multi-threading. OpenMP supports scheduling and load balancing of threads implicitly at the system level. Both of these help achieve parallelism on a multicore machine. Pthreads use low-level APIs and have fine-grain control. On the other hand, OpenMP uses high-level pragma directives that are portable, scalable, and gives programmers a simple and flexible interface [25]. A few of our observations:

We use the *nowait* clause of OpenMP parallel pragma to achieve similar functionality of the no-sync PageRank variant and use a default *barrier* clause for pragmas at the end to achieve blocking synchronization. We tested our Barriers and No_Sync parallel variants with varied thread count on synthetic and web-graphs standard datasets using Pthreads and OpenMP APIs.

From Figs. 7a, 7b and 8a, b, it is observed that the speed-up obtained is almost identical for both Pthreads and OpenMP implementations, with Pthreads being

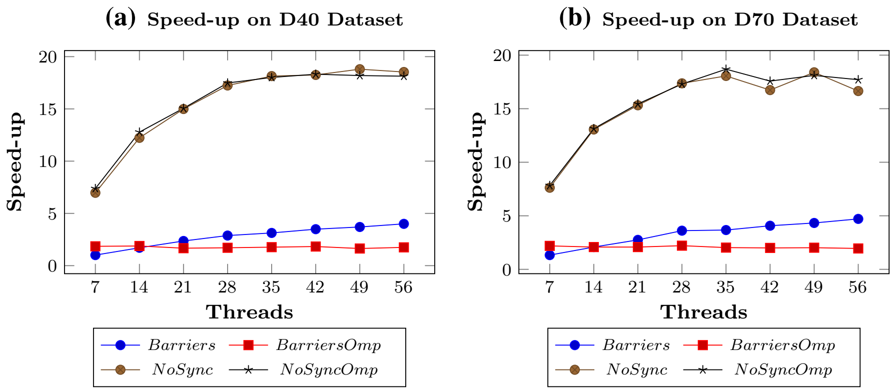


Fig. 7 Pthreads vs. OpenMP analysis of Speed-up computations on Synthetic Datasets

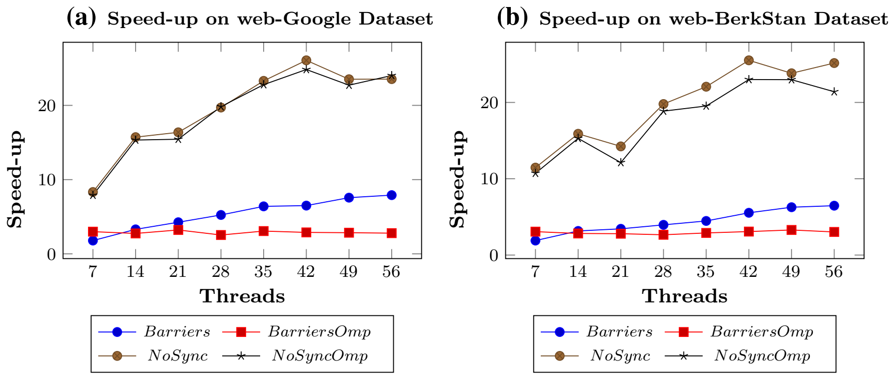


Fig. 8 Pthreads vs. OpenMP analysis of Speed-up computations on Standard Datasets

slightly better performant. It is also observed that the No_Sync variant performs better than the Barriers variant with Pthreads and OpenMP implementations.

Asynchronous methods with non-blocking progress properties like lock-freedom and wait-freedom can be achieved by using Pthreads. In this work, we demonstrated that using Pthreads lock-freedom can be achieved for computing PageRank. But it is not clear how to implement non-blocking method s using OpenMP. This is because with Pthreads, we can specify the running code of each threads which is not possible with OpenMP.

6 Conclusion and Future work

This paper proposed a Non-Blocking (No-Sync and Wait-free) algorithms to implement the parallel PageRank algorithm on Shared Memory architectures. The proposed methods replace the Lock-Based and Barrier synchronization mechanism found in the state-of-the-art approaches. Our simulation results on various graphs found that our approach will achieve better performance when combined with the existing methods. The results shown in this paper motivates that the non-blocking variants, when applied for iterative algorithms, can lead to performance improvements.

As part of future work, we plan to integrate our proposed approach with the existing graph framework. We also plan to apply our approaches for applications where iterative algorithms are the direction for future work. The source code is available on[1].

References

1. Gross, J., Yellen, J.: Graph Theory and Its Applications. CRC Press, Inc., Boca Raton (1999)
2. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, previous number = SIDL-WP-1999-0120. Stanford InfoLab (1999). http://ilpubs.stanford.edu:8090/422/
3. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 International Conference on Management of Data, New York, NY, USA, pp 135–146 (2010). https://doi.org/10.1145/1807167.1807184
4. Roy, A., Mihailovic, I., Zwaenepoel, W.: X-Stream: edge-centric graph processing using streaming partitions. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, pp. 472–488. Association for Computing Machinery, New York (2013). https://doi.org/10.1145/2517349.2522740
5. Hong, S., Chafi, H., Sedlar, E., Olukotun, K.: Green-Marl: a DSL for easy and efficient graph analysis. In: ASPLOS XVII, pp. 349–362. Association for Computing Machinery, New York (2012). https://doi.org/10.1145/2150976.2151013
6. Sidiroglou-Douskos, S., Misailovic, S., Hoffmann, H., Rinard, M.: Managing performance vs. accuracy trade-offs with loop perforation. In: ESEC/FSE '11, pp. 124–134. Association for Computing Machinery, New York (2011). https://doi.org/10.1145/2025113.2025133
7. Panyala, A., Subasi, O., Halappanavar, M., Kalyanaraman, A., Chavarr,ía-Miranda, D.G., Krishnamoorthy, S.: Approximate computing techniques for iterative graph algorithms. In: 24th IEEE International Conference on High Performance Computing, HiPC 2017, Jaipur, India, 18–21 December 2017, pp 23–32. IEEE Computer Society (2017). https://doi.org/10.1109/HiPC.2017.00013
8. Vu, J.: The art of multiprocessor programming by Maurice Herlihy and Nir Shavit. ACM SIGSOFT Softw. Eng. Notes **36**(5), 52–53 (2011). https://doi.org/10.1145/2020976.2021006
9. Shun, J., Blelloch, G.E.: Ligra: a lightweight graph processing framework for shared memory. In: Nicolau, A., Shen, X., Amarasinghe, S.P., Vuduc, R.W. (eds) ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, 23–27 February 2013, pp. 135–146. ACM (2013). https://doi.org/10.1145/2442516.2442530
10. Garg, P., Kothapalli, K.: STIC-D: algorithmic techniques for efficient parallel PageRank computation on real-world graphs. In: Proceedings of the 17th International Conference on Distributed Computing and Networking, Singapore, 4–7 January 2016, pp. 15:1–15:10. ACM (2016). https://doi.org/10.1145/2833312.2833322
11. Lakhotia, K., Kannan, R., Pati, S., Prasanna, V.K.: GPOP: a scalable cache- and memory-efficient framework for graph processing over parts. ACM Trans. Parallel Comput. **7**(1), 7:1-7:24 (2020). https://doi.org/10.1145/3380942
12. Barrett, B.W., Berry, J.W., Murphy, R.C., Wheeler, K.B.: Implementing a portable multi-threaded graph library: the MTGL on Qthreads. In: 23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, 23–29 May 2009, pp. 1–8. IEEE (2009). https://doi.org/10.1109/IPDPS.2009.5161102

[1] https://github.com/PDCRL/PageRank.

13. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: GraphLab: a new framework for parallel machine learning. In: Grünwald, P., Spirtes, P. (eds) UAI 2010, Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence, Catalina Island, CA, USA, 8–11 July 2010, pp. 340–349. AUAI Press (2010). https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=2126&proceeding_id=26
14. Mitliagkas, I., Borokhovich, M., Dimakis, A.G., Caramanis, C.: FrogWild!—fast PageRank approximations on graph engines. CoRR abs/1502.04281 (2015). arxiv:1502.04281
15. Wang, G., Xie, W., Demers, A.J., Gehrke, J.: Asynchronous large-scale graph processing made easy. In: CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Online Proceedings, Asilomar, CA, USA, 6–9 January 2013 (2013). www.cidrdb.org, http://cidrdb.org/cidr2013/Papers/CIDR13_Paper58.pdf
16. Nguyen, D., Lenharth, A., Pingali, K.: A lightweight infrastructure for graph analytics. In: Kaminsky, M., Dahlin, M. (eds) ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, 3–6 November 2013, pp. 456–471. ACM (2013). https://doi.org/10.1145/2517349.2522739
17. Beamer, S., Asanovic, K., Patterson, D.A.: Reducing PageRank communication via propagation blocking. In: 2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, 29 May–2 June 2017, pp. 820–831. IEEE Computer Society (2017). https://doi.org/10.1109/IPDPS.2017.112
18. Omar, H., Ahmad, M., Khan, O.: GraphTuner: an input dependence aware loop perforation scheme for efficient execution of approximated graph algorithms. In: 2017 IEEE International Conference on Computer Design, ICCD 2017, Boston, MA, USA, 5–8 November 2017, pp. 201–208. IEEE Computer Society (2017). https://doi.org/10.1109/ICCD.2017.38
19. Peng, Z., Powell, A., Wu, B., Bicer, T., Ren, B.: GraphPhi: efficient parallel graph processing on emerging throughput-oriented architectures. In: Evripidou, S., Stenström, P., O'Boyle, M.F.P. (eds) Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT 2018, Limassol, Cyprus, 1–4 November 2018, pp. 9:1–9:14. ACM (2018). https://doi.org/10.1145/3243176.3243205
20. Eedi, H., Peri, S., Ranabothu, N., Utkoor, R.: An efficient practical non-blocking PageRank algorithm for large scale graphs. In: 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2021, Valladolid, Spain, 10–12 March 2021, pp. 35–43. IEEE (2021). https://doi.org/10.1109/PDP52278.2021.00015
21. Luo, L., Liu, Y.: Processing graphs with barrierless asynchronous parallel model on shared-memory systems. Future Gener. Comput. Syst. **106**, 641–652 (2020). https://doi.org/10.1016/j.future.2020.01.033
22. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: a recursive model for graph mining. In: Berry, M.W., Dayal, U., Kamath, C., Skillicorn, D.B. (eds) Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, 22–24 April 2004, pp. 442–446. SIAM (2004). https://doi.org/10.1137/1.9781611972740.43
23. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection (2014). http://snap.stanford.edu/data
24. Rossi, R.A., Ahmed, N.K.: The network data repository with interactive graph analytics and visualization. In: AAAI (2015). http://networkrepository.com
25. Kumar, V., Grama, A., Gupta, A., Karypis, G.: Introduction to Parallel Computing: Design and Analysis of Algorithms. Benjamin-Cummings Publishing Co., Inc., San Francisco (1994)

Authors and Affiliations

Hemalatha Eedi[1] · **Sahith Karra**[2] · **Sathya Peri**[1] · **Neha Ranabothu**[1] · **Rahul Utkoor**[1]

✉ Hemalatha Eedi
cs15resch11002@iith.ac.in

Sahith Karra
sahith.karra@gmail.com

Sathya Peri
sathya_p@cse.iith.ac.in

Neha Ranabothu
cs14btech11028@iith.ac.in

Rahul Utkoor
cs14btech11037@iith.ac.in

[1] Indian Institute of Technology Hyderabad, Hyderabad, Telangana, India

[2] American High School, Fremont, CA, USA