An Efficient Practical Non-Blocking PageRank Algorithm for Large Scale Graphs*

Hemalatha Eedi*, Sathya Peri†, Neha Ranabothu‡, Rahul Utkoor§
Department of Computer Science and Engineering
Indian Institute of Technology Hyderabad, Telangana State, India 502285
*Email: *cs15resch11002@iith.ac.in*
†*sathya_p@cse.iith.ac.in,* ‡*cs14btech11028@iith.ac.in,* §*cs14btech11037@iith.ac.in*

Abstract—**PageRank algorithm is a benchmark for many graph analytics and is the underlying kernel for link predictions, recommendation systems. It is an iterative algorithm that updates ranks of pages until the value converges. Implementation of PageRank algorithm on a shared memory architecture while taking advantage of fine-grained parallelism using large-scale graphs is a challenging task. In this paper, We present parallel algorithms for computing the PageRank suitable to the shared memory systems. Initially, we present parallel implementations of page-rank algorithms using barrier and lock variants. Later, we propose new approaches which are lock-free and are barrier-less synchronization to overcome the issues of lock based methods.**

A detailed experimental analysis of our approach is carried out using real-world web graphs from SNAP and Synthetic Graphs from RMAT on an Intel(R) Xeon E5-2660 v4 processor architecture with 56 threads using the POSIX thread library.

Index Terms—**Shared Memory Architecture, PageRank Algorithm, Fine-Grained Parallelism, No Synchronization and Wait-Free.**

1. Introduction

Graphs have ubiquitous for representing data in various fields such as Biology, Genomics, Astrophysics, Transportation Networks, Web and Social Network Analysis, Scientific Computing [1]. In general many of these graphs are huge and scale with billions of nodes and edges with irregular and intricate structures. As a result, there have been several efforts for developing graph frameworks, and graph libraries to address these issues.

Performance is still a big issue in processing graphs and graph applications, especially in shared memory architectures. On many of these large graphs, it is also required to exploit the nature and understanding of these graphs by applying some metrics for deriving meaningful analytics. The PageRank is one such property to find the quality of nodes in a web graph. Page et al. [2] devised this algorithm for Google Search Engine. The PageRank computation proceeds iteratively over and over again to estimate the significance of a web page. The primary understanding of the algorithm derives the rank of a page based on its incoming channels. If the incoming link is from a highly ranked page, it is evident that the current page also has a high rank [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 = number of pages, q = outdegree defining the number of hyperlinks on page v and d is the dampening parameter initialized to 0.85.

The first part of the equation denotes a random probability of reaching a node v from some other point. In this case, d represents the probability of selecting the node u from a complete set of n nodes. The second part of the equation describes the factors contributed by all incoming nodes arriving at a node v. The summation of the PageRank values from the incoming nodes of v is multiplied by the learning factor d.

The convergence of PageRank iterations and addressing dangling vertices are primary concerns in computing PageRank. The convergence of PageRank is guaranteed only provided the Web graph is strongly connected and is aperiodic [3].

Pages with no outgoing links must also be resolved to patch the computation issues of PageRank. Because there is no possible distribution of ranks between other vertices, these pages act as a rank sink. As pendant vertices do not affect other vertices' page rank, in a study, it was formally proposed to iteratively drop the vertices with no out-links from the graph to resolve this computational issue [2].

PageRank is an iterative algorithm. In each step, the algorithm approximates the order of page ranks till it reaches the solution. The rank of the node implies its importance in the graph. Computing PageRank of a single node depends on the incoming edges and out-degree of the respective node.

Non-blocking (lock-freedom or wait-freedom) is an important progress property, which ensures that the system makes progress regardless of how some thread in the system has become slow or even has crashed [4]. As a result, to achieve non-blocking progress, the threads executing cannot acquire locks since a thread that has acquired a lock can po-

This work is supported by Intel Grant: "Tools for Large-Scale Graph Analytics"

tentially block other threads from progressing by becoming slow or crashing.

Achieving *non-blocking* progress on iterative algorithms is a challenging task [5]. In this work, we enabled Page Rank iterative algorithm to achieve lock-freedom and wait-freedom property.

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. Till now, barrier based solutions are considered to be useful for achieving better parallelism on iterative algorithms. However, these solutions do not guarantee non-blocking progress guarantees lock-freedom/wait-freedom properties. Spotting independence across the iterations of the page-rank algorithm is non-trivial. Till date, the approaches proposed for achieving better parallelism on the page-rank algorithm focuses on graph-optimization/adjacency-matrix optimization* techniques. Our technique is unique, which guarantees non-blocking progress property on a page-rank algorithm. We used piece-wise concurrent programming by removing barrier constraint from an iterative algorithm and eliminates the dependency between the iterations.

1.1. System Model

In this work, we assume that our system consists of *p* threads running on multiprocessors. These threads are logically divided into partitions and are assigned to a specific processor. Threads in each partition can use shared local memory and communicate using thread APIs. To deal with the issues raised during thread communication, we implement atomic primitives - CAS(Compare-And-Swap).

1.2. Formal Definitions

This section gives a formal description of synchronization approaches [4] for designing shared data objects and algorithms used in this paper.

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.

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. The lock-free approach ensures that infinitely often, some thread finishes in a finite number of steps. To implement synchronization using lock-free and wait-free approaches, the most prominent atomic primitive used is Compare-And-Swap. [4]

2. Related Work

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], [6], [7], [8], to mention a few.

Parallel PageRank algorithm proposed by Berry er al. [6], in their Multi-Threaded Graph Library (MTGL), runs on Cray XMT (Multi-Threaded Architecture extended with 128 threads) used QThreads 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. [9] evolved into distributed systems for implementing parallel machine learning algorithms. 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. [10] However, in each iteration, the PageRank computations in parallel are carried out by using synchronization locks and barriers.

Wang et al. [11] in their paper titled Asynchronous Large-Scale Graph Processing Made Easy, 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.

Galois is a vertex-centric data-parallel programming model on shared memory systems, proposed by Nguyen et al. [12], that uses a coordinated scheduling approach where the computations are breakdown into supersteps. At each superstep, a barrier synchronization is invoked. Galois does not satisfy the lock-free and wait-free properties.

Parallel PageRank algorithm implemented using Ligra proposed by Shun and Blelloch [7] uses simple routines. It takes advantage of Frontier Based computations where an active set of vertices and edges are dynamically changing through the duration of execution. To achieve parallelism, Ligra uses Clik Plus parallel codes.

Garg et al. [8] 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 exploits the nature of real-world graphs and reduces the PageRank computation by removing redundancies in edges and nodes of the graph. This kind 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 [13] 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.

Ajay Panyala et al. [14] presents several approximate computing techniques like loop perforation and data caching to improve the PageRank algorithm's performance. The performance results show 7-10 times better improvement when compared with an efficient algorithm STICD [8]. 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.

GraphPhi framework proposed by Zhen Peng et al. [15] 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.

3. Description of Algorithms

Most of the research on PageRank computation is on the pre-processing step, i.e., processing the graph (STICD, Ligra) [7] [8], equal distribution of load to the threads [16], etc. In the end, all these algorithms use a Barrier synchronization after each iteration. Using a Barrier has drawbacks as every thread needs to wait at each iteration without making any progress. Our main motive is to increase the computational speed by avoiding barriers and allowing the threads to run independently throughout the execution. In this section, we explain the baseline parallel algorithm using Barrier synchronization. Later we propose Blocking and Non-Blocking algorithms by eliminating the Barrier synchronization.

3.1. Baseline Barrier Synchronization Algorithm

Given a graph *G = (V, E)* is partitioned into *p* equal-sized partitions where *p* is the number of threads and partition P_i contains all the vertices with id $\in \{i{*}k | k \in (1, n/p)\ and\ i \in (1, p)\}$. The barrier synchronization algorithm given in Algorithm 1 starts with initializing error, threshold, and initial PageRank values of all the graph nodes. The parallel algorithm (at Line 3) then computes the PageRank of nodes by creating *p* number of threads. Each thread is responsible for computing the new PageRank value of its allocation P_i by reading and summing up the previous PageRank values of its incoming neighbor nodes. *pr* and *prPrev* variables store the PageRank values of all vertices from current and previous iteration. Each thread stores the error obtained from its allocation to a thread local variable *thErr[T_i]*. After every iteration the main thread computes the max error from all the *thErr* variables. This process iterates in parallel until it converges ($error < threshold$).

Algorithm 1 Using Barrier

```
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:     $threshold \leftarrow \frac{0.01}{\#ofVertices}$
6:     for all nodes $u_i \mid i \in \{1, ..., n\}$ do
7:         $pr(u_i) \leftarrow 0$
8:         $prPrev(u_i) \leftarrow \frac{1}{n}$
9:     end for
10:    $thrErr[p] \leftarrow 0$                        ▷ Initialize thread's error
11:    while $error <\ threshold$ do
12:        for all threads $T_i \mid i \in \{1, ..., p\}$ do
13:            for all $u \in threadVertices(T_i)$ do
14:                $pr(u) \leftarrow \frac{(1-d)}{n}$
15:                for all $u \in V$ such that (v,u) ∈ E do
16:                    $pr(u) = pr(u) + \frac{prPrev(v)}{outDeg(v)} * d$
17:                end for
18:                $thrErr[T_i] = max(thrErr[T_i], |prPrev(u) - pr(u)|)$
19:            end for
20:        end for
21:        Barrier checkpoint
22:        for all threads $T_i \mid i \in \{1, ..., p\}$ do    ▷ Update Global Error
23:            $error = max(error, thrErr[T_i])$
24:        end for
25:        $prPrev = pr$
26:        Barrier checkpoint
27:    end while
28: end procedure
```

3.2. Lock-based PageRank Algorithm

In the lock-based algorithm, we eliminate the previous PageRank variable *(prPrev)* and allow the threads to read-write to the same single PageRank variable. Locks avoid read-write conflicts that arise when threads are making progress in different iterations simultaneously. For example, *thd1* is trying to compute the page rank of node *u* in kth iteration by accessing the PageRank of *v* in (k-1)th iteration where *(u,v)*$\in$ *E*. There is a possibility that *v* belongs to *thd2* and its (k-1)th iteration is updated at the same time. We eliminate these read-write conflicts by introducing locks. The vertices in each partition are classified into *Boundary vertices* and *Internal Vertices* to reduce the time spent in attaining the locks. Boundary vertices are those vertices for which at least one incoming neighbor vertex belongs to other partitions. Internal vertices are those vertices for which all incoming neighbor vertices belong to the same partition. This partition of vertices into internal and boundary is helpful as there won't be any conflicts while computing the

PageRank of internal vertices. We propose two variant of fine-grained locking methods to attain locks on boundary vertices.

3.2.1. Fine-Grained-Lock-Variant1. In the locking algorithm, we attain locks on all incoming neighbor vertices of the node before computing the PageRank. In each iteration and for each node that belongs to *Boundary Vertices*, the thread acquires the each lock on all its incoming neighboring nodes, computes the PageRank, updates the newly computed PageRank value by accessing shared values inside its critical section and release the locks. The vertices are always locked in the increasing order of their Ids to avoid deadlock-free. The selected node is also locked in the order along with its incoming neighbor vertices. Similarly, release the locks in the decreasing order of their Ids.

Algorithm 2 Algorithm Fine-Grained-Lock-Variant1 Pagerank(G)

```
1: procedure ComputePR(node u)
2:     $prev \leftarrow pr(u)$
3:     $temp = \frac{(1-d)}{n}$
4:     for all $v \in V$ such that $(v, u) \in E$ do
5:         $temp = temp + \frac{pr(v)}{outDeg(v)} * d$
6:     end for
7:     $thrlocErr = max(thrlocErr, |temp - prev|)$
8:     return temp
9: end procedure
10: procedure Main(Graph G = (V,E))
11:     G = CovertCsr(V,E)
12:     internalVertices, boundaryVertices = CategorizeVertices(V,E)
13:     Initialize Variables
14:     for all $T_i \in threads(1, p)$ do
15:         while $localError < threshold$ do
16:             $thrlocErr \leftarrow 0$
17:             for all $u \in internalVertices(T_i)$ do
18:                 $pr(u) \leftarrow ComputePR(u)$
19:             end for
20:             for all $u \in boundaryVertices(T_i)$ do
21:                 for all $v \in V$ such that $(v, u) \in E \cup u$ do
22:                     $v.lock()$
23:                 end for
24:                 $pr(u) \leftarrow ComputePR(u)$
25:                 for all $v \in V$ such that $(v, u) \in E \cup u$ do
26:                     $v.unlock()$
27:                 end for
28:             end for
29:             glb.lock()
30:             $thErr[T_i] \leftarrow thrlocErr$
31:             $localError \leftarrow 0$
32:             for all $tid \in threads(1, p)$ do
33:                 $localError \leftarrow max(localError, thErr[tid].load())$
34:             end for
35:             glb.unlock()
36:         end while
37:     end for
38: end procedure
```

3.2.2. Fine-Grained-Lock-Variant2. In the fine-grained Locking mechanism, instead of locking all its incoming nodes at once each thread locks the incoming node one after the other, reads the value, adds the sum to its local variable and it releases the lock. The same process continues for all the other incoming neighbour nodes in its partition. In this algorithm there won't be any problem of deadlock as the threads are not allowed to attain more than one lock at the same time.

Algorithm 3 Algorithm Fine-Grained-Lock-Variant2 Pagerank(G)

```
1: procedure ComputePRInternal(node u)
2:     $prev \leftarrow pr(u)$
3:     $temp = \frac{(1-d)}{n}$
4:     for all $v \in V$ such that $(v, u) \in E$ do
5:         $temp = temp + \frac{pr(v)}{outDeg(v)} * d$
6:     end for
7:     $thrlocErr = max(thrlocErr, |temp - prev|)$
8:     return temp
9: end procedure
10: procedure ComputePRBoundary(node u)
11:     $prev \leftarrow pr(u)$
12:     $temp = \frac{(1-d)}{n}$
13:     for all $v \in V$ such that $(v, u) \in E$ do
14:         v.lock()
15:         $temp = temp + \frac{pr(v)}{outDeg(v)} * d$
16:         v.unlock()
17:     end for
18:     $thrlocErr = max(thrlocErr, |temp - prev|)$
19:     return temp
20: end procedure
21: procedure Main(Graph G = (V,E))
22:     G = CovertCsr(V,E)
23:     internalVertices, boundaryVertices = CategorizeVertices(V,E)
24:     Initialize Variables
25:     for all $T_i \in threads(1, p)$ do
26:         while $localError < threshold$ do
27:             $thrlocErr \leftarrow 0$
28:             for all $u \in internalVertices(T_i)$ do
29:                 $pr(u) \leftarrow ComputePRInteral(u)$
30:             end for
31:             for all $u \in boundaryVertices(T_i)$ do
32:                 $pr(u) \leftarrow ComputePRBoundary(u)$
33:             end for
34:             glb.lock()
35:             $thErr[T_i] \leftarrow thrlocErr$
36:             $localError \leftarrow 0$
37:             for all $tid \in threads(1, p)$ do
38:                 $localError \leftarrow max(localError, thErr[tid].load())$
39:             end for
40:             glb.unlock()
41:         end while
42:     end for
43: end procedure
```

3.3. Non-Blocking Algorithm

Lock-based algorithms are easy to implement. When a thread holding a lock cannot make progress, other threads waiting for a long time may not progress and lead to starvation or deadlock. Non-blocking algorithms are alternatives to lock-based algorithms that are immune to starvation and deadlock problems. Non-blocking algorithms use low-level machine instructions such as CAS (Compare-And-Swap) to guarantee data consistency and correctness under concurrent shared data access.

3.3.1. No-Sync . In this Non Synchronization algorithm, at least one thread should compute and update the PageRank values of its partition. Each node's PageRank variable is updated using atomic CAS operations to attain the No-Sync property.

In Algorithm 4, from lines 11 to 26 are executed in parallel by all threads. We store the PageRank value associated with each node in atomic variables. The update of the PageRank value of a node is done on line 17 using CAS operations. Each thread updates the local variable *localError* with the max value from global variable *thErr* from line 22 to 24 and uses it to check if it can proceed to the next iteration in line 12. The update of *localError* is done after computing of pagerank of all nodes in its own partition (i.e) single iteration of the thread.

Algorithm 4 Algorithm No-Sync Pagerank(G)

```
1: procedure COMPUTEPR(node u)
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
8: procedure MAIN(Graph G = (V,E))
9:     G = CovertCsr(V,E)
10:    Initialize Variables
11:    for all $T_i \in threads(1,p)$ do
12:        while $localError < threshold$ do
13:            $thrlocErr \leftarrow 0$
14:            for all $u \in threadVertices(T_i)$ do
15:                $prev \leftarrow pr(u)$
16:                $temp \leftarrow ComputePR(u)$
17:                $pr(u).store(temp)$
18:                $thrlocErr = max(thrlocErr, |temp - prev|)$
19:            end for
20:            $thErr[T_i].store(thrlocErr)$
21:            $localError \leftarrow 0$
22:            for all $tid \in threads(1,p)$ do
23:                $localError = max(localError, thErr[tid].load())$
24:            end for
25:        end while
26:    end for
27: 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 ith iteration is given by Eq(1) and Eq(2) 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. Eq(1) can be modified to Eq(3) considering that the threads are present in ith and (i-1)th iterations. Let S_i^u be a set of vertices where (v,u) ∈ E and PageRank of v is from ith iteration.

$$pr_{i-1:i}^u = \frac{1-d}{n} + d* \sum_{v\in S_i^u} \frac{pr_{i-1}^v}{outDeg(v)} + d* \sum_{v\in S_{i-1}^u} \frac{pr_{i-2}^v}{outDeg(v)} \tag{4}$$

Error in Eq(2) can also be modified accordingly.

$$err_{i-1:i}^u = \left| pr_{i-1:i}^u - pr_{i-1}^u \right| \tag{5}$$

At any given instant $pr_{i-1}^u \leq pr_{i-1:i}^u \leq pr_i^u$ if $pr_{i-1}^u \leq pr_i^u$ which means $pr_{i-1:i}^u$ always lies between pr_i^u and pr_{i-1}^u.

$$\left| pr_{i-1:i}^u - pr_{i-1}^u \right| \leq \left| pr_i^u - pr_{i-1}^u \right| \Rightarrow err_{i-1:i}^u \leq err_i^u \tag{6}$$

err_i^u from the base algorithm is always expected to decrease in every iteration, so $err_{i-1:i}^u$ 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. Eq(3) can be modified for the threads to be present in 1st to Ith iteration.

$$\widehat{pr_i^u} = \frac{1-d}{n} + \ d * \sum_{l=1}^{I} \sum_{v\in S_l^u} \frac{\widehat{pr_l^v}}{outDeg(v)} \tag{7}$$

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(6) can be modified as Eq(7) where $S^u = \bigcup_{l=1}^{I} S_l^u = \{v|(v,u) \in E\}$.

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

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. Eq(7) is exactly same as Eq(1) 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. □

3.3.2. Wait-free.

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 the PageRank computed for all nodes in that particular iteration. Any thread which finishes the computation of its allocation will help any other random thread to complete its assignment before proceeding into the next iteration. The threads continue to help other threads in progress until PageRank of all nodes gets computed.

```
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;
};
```

In algorithm 5, all available threads executed 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 is done from lines 56 to 59. Each thread has a global atomic variable (glbThdInfo), which stores the info 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* updates the information in the *Thd2* global variable. This update of *GlbThdInfo* is done from lines 24 to 28 in ComputePR( ) procedure. UpdatePR( ) method from lines 1 to 12 is used to update the PageRank value along with 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, as some helper thread would already do the latest update.

3.4. STICD

We extended our proposed approaches by considering two optimizations from paper [8] - Identical Nodes and Chain Nodes.

To show our implementation's scalability and robustness, we extended the No-Sync variant to support Identical nodes and Chain nodes optimizations defined in the paper and named them No-Sync-STICD-IN and No-Sync-STICD-CN respectively. In paper [8], the authors have shown the results on four different datasets based on the graph's properties. We have evaluated our algorithms on some real-time datasets [17], [18] to show performance improvement compared with our proposed methods.

Algorithm 5 Wait-Free

```
1: procedure UPDATEPR(u, nodePr, thdVar)
2:     $z \leftarrow pr(u)$
3:     if $z.itr == thdVar.itr$ then
4:         $casOb \leftarrow newPrCasOb(thVar.itr++, nodePr)$
5:         $CAS(pr(u), z, casOb)$
6:     end if
7:     $z \leftarrow prevPr(u)$
8:     if $z.itr == thdVar.itr$ then
9:         $casOb \leftarrow newPrCasOb(thVar.itr++, nodePr)$
10:        $CAS(prevPr[u], z, casOb)$
11:    end if
12: end procedure
13: procedure COMPUTEPR(thdId, hlpId, thdVar)
14:    $thInfo \leftarrow glbThInfo[hlpId].load()$
15:    while $u \in ThreadVertices$ and $glbVar.itr == thdVar.itr$ do
16:        $nodePr \leftarrow \frac{(1-d)}{n}$
17:        for all $v \in V$ such that $(v, u) \in E$ do
18:            $nodePr+ = \dfrac{glbPrevPr[v]}{outDeg(v)} * d$
19:        end for
20:        Invoke updatePr(u,nodePr,thdVar)
21:        $z = glbThInfo[hlpId]$
22:        if $z.itr == thdVar.itr$ then
23:            $er \leftarrow max(z.er, |nodePr - prevPr|)$
24:            $casOb \leftarrow newThCASOb(z.itr, next(u, hepId), er)$
25:            $CAS(glbThInfo[hlpId], z, casOb)$
26:        end if
27:    end while
28: end procedure
29: procedure UPDATEGLBVAR((thId,hlpId,thdVar)
30:    while $true$ do
31:        $z \leftarrow glbVar$
32:        if $z.itr == thdVar.itr$ then
33:            $casOb \leftarrow copy(z)$
34:            $casOb.check[helpId] \leftarrow true$
35:            $casOb.er \leftarrow max(casOb.er, glbThInfo[hlpId].er)$
36:            $CAS(glbVar, z, casOb)break$
37:        end if
38:    end while
39:    while $true$ do
40:        $z \leftarrow glbThInfo[hlpId]$
41:        if $z.itr == thdVar.itr$ then
42:            $casOb \leftarrow newThCASOb(z.itr++, thdId, 0)$
43:            $CAS(glbThInfo[hlpId], z, casOb)break$
44:        end if
45:    end while
46: end procedure
47: procedure THREADPAGERANK(thdId)
48:    while $glbVar.load().er > threshold$ do
49:        Invoke ComputePR(thdId,thdId,thdVar)
50:        for all $thr \in threads$ and $thr! = thdId$ and notCompletePR(thr) do
51:            Invoke ComputePR(thr,thdId,thdVar)
52:        end for
53:        Intialize error value to 0 for next iteration in glbVar using CAS
54:        Invoke UpdateGlbVar(thdId,thdId,thdVar)
55:        for all $thr \in threads$ and $thr! = thdId$ and notCompleteGlbVar(thr) do
56:            Invoke UpdateGlbVar(thr,thdId,thdVar)
57:        end for
58:        Intialize itr in glbVar using CAS
59:        $thdVar \leftarrow glbVar.load()$
60:        Invoke Swap()
61:    end while
62: end procedure
```

4. Experiments Evaluation

4.1. Platform

We conducted our simulations on a 56 core Intel(R) Xeon(R) E5-2660 v4 processor architecture running at 2.06 GHz core frequency. Each core supports two logical threads and two CPU socket(s) with 14 cores per socket. Every core's L1 - 32K, L2 - 256K cache memory is private to that core, and L3 - 35840K cache memory is shared across the cores . All the simulations were coded in C/C++ and compiled using g++ 7.5.0 and using the POSIX Multi-Threaded library.

4.2. Datasets

We use 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 [7], [8], [19] in providing a fair comparison. We conduct initial experiments on randomly generated synthetic graphs in the range of $1 * 10^6 to 7 * 10^6$ with *n* vertices and *(n-1)* edges. Later on, Web-Graphs, Social-Networks, Road-Networks, and Collaboration-Networks from standard datasets repository [17], and three synthetic RMAT graph datasets [20] with a scale-factor of 21 ,and 22 the existing studies. All the [19] graph datasets sizes range from 0.05GB to 2.5GB and are in Adjacency List format, which is later converted to CSR (Compressed Sparse Row) format. We tested all the proposed algorithms on given datasets.

TABLE 1. REAL-WORLD AND SYNTHETIC GRAPH DATASETS

Datasets	#vertices	#Edges	Size
Web Graphs [17]			
web-Stanford	281903	2312497	30 MB
web-NotreDame	325729	1497134	20 MB
web-BerkStan	685230	7600595	20 MB
web-Google	875713	5105039	7 MB
Social Networks [18]			
soc-Epinions1	75879	508837	5.7 MB
Slashdot0811	77360	905468	10.7 MB
Slashdot0902	82168	948464	11.3 MB
soc-LiveJournal1	4847571	68993773	1100 MB
Road Networks [18]			
road-italy-osm	6686493	7013978	109.9 MB
great-britain-osm	7.7M	8.2M	28 MB
asia-osm	12M	12.7M	5.1 MB
germany-osm	11.5M	12.4M	98.5 MB
Collaborative Network [18]			
co-AuthorsCiteseer	227.3K	814.1K	10.1 MB
ca-coauthors-dblp	540.5K	15.2M	200.6 MB
RMAT Graphs [20] [19]			
RMAT_21	2097152	41943040	565.4 MB
RMAT_22	4194304	83886080	1.2GB

4.3. Results and Discussion

In this section we present the speedups achieved by parallel variants of Pagerank algorithms. In this section, we present the speed-up achieved by parallel variants of PageRank algorithms. The ratio between Sequential execution time and Parallel execution time is the metric for calculating the algorithm's speed-up. With a fixed number of threads(32) on a different class of datasets, we execute the programs and obtain the execution times. When incorporated with existing graph processing methods, the proposed algorithms prove prominence improvement at the hardware level.

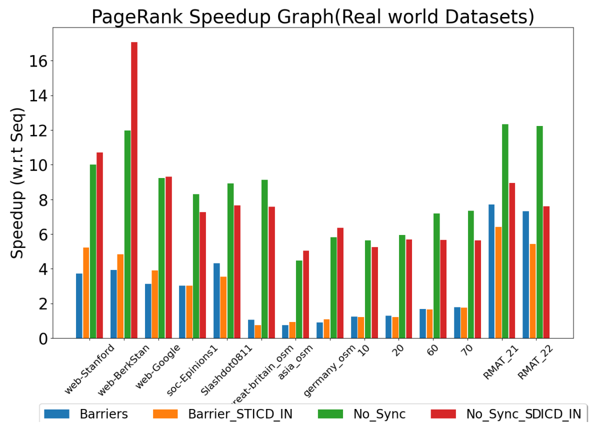

Figure 1. Speed-Up on Real-World Datasets

The results of the speed-up of the Barrier and the No-Sync implementations upon 32 threads are shown in in Figure 1, Figure 2 and Figure 3. We observed that No-Sync and No-Sync-STICD algorithms perform better than other variants for almost all the datasets. For the first three datasets (Stanford, BerkStan, and Epinions1), Barriers-STICD-IN performs better than Barriers, and the addition of our No-Sync algorithm to STICD-IN further increases the performance. The performance improvements proved are due to the piecewise parallel implementation of non-blocking algorithms using atomic primitives. For Livejournal1 and coauthors-dblp datasets, Barriers-STICD-IN is incompetent than Barriers.

In Figure 2 and Figure 3, we show the speed-up obtained between the parallel variants of Pagerank algorithms on a few Standard Real-World and Synthetic Datasets respectively. The datasets were selected in such a way as to compare the performance of STICD and STICD-CN . No-Sync and No-Sync-STICD-CN algorithms are performing better than other variants for almost all the datasets. Similar to Figure1, the performance trend between No-Sync and No-Sync-STICD-CN is precisely the same as that of Barriers and Barriers-STICD-CN . On average, No-Sync has a speed-up of around 2.7 w.r.t base algorithm because of the elimination of barrier constraint.

In Figure 4, we show the speed-ups obtained by parallel variants by varying the threads on a fixed dataset(RMAT22).

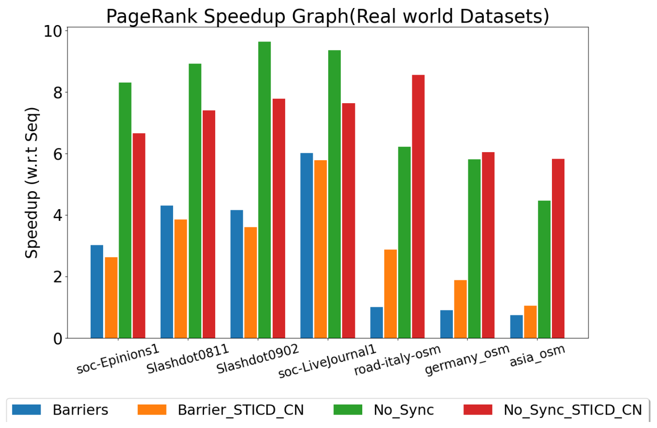

Figure 2. Speed-Up on Real-world Graphs

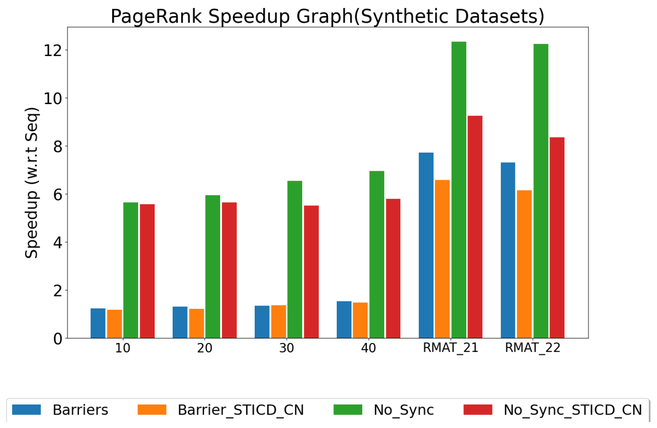

Figure 3. Speed-Up on Synthetic Graphs

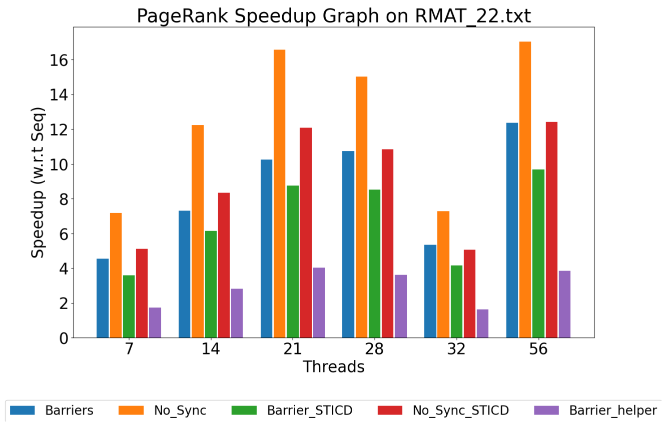

Figure 4. Speed-Up w.r.t RMAT_22 Dataset

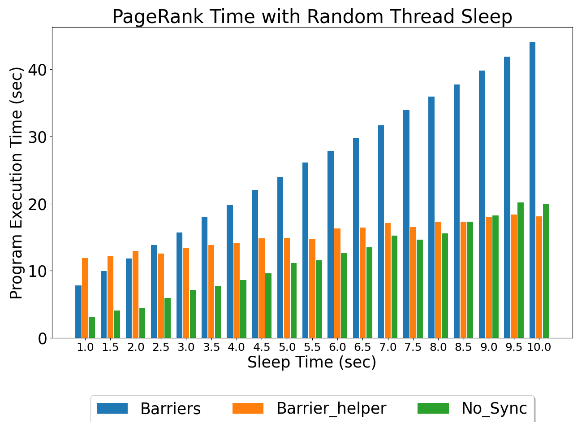

Figure 5. Simulation results for the Sleeper Threads - Plot5

In this work, we are applying the static load balancing technique in all parallel variants. Till 14 threads, we can see the gradual increase in the speed-up as we increase the threads. After a point, increasing the threads will not give a better speed-up. We are achieving maximum parallelism at 14 threads.

Sleeping variants: To evaluate the impact of Wait-free algorithm, we deterministically added sleep to the threads in selected iterations. In the case of Barrier algorithm, each thread has to wait until the completion of the sleeping thread. In the case of No-sync, the work corresponds to the sleeping thread will be resumed after 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 other thread and helps other threads after completing the task assigned to it. In Figure 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.

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

5. Conclusion and Future Work

This paper proposed a No Synchronization and a Wait-free synchronization mechanism to implement a 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

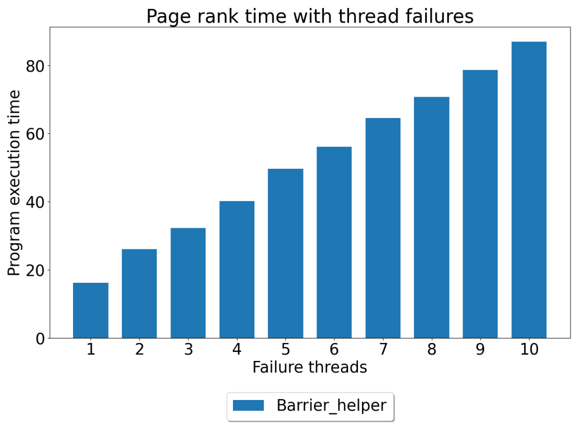

Figure 6. Simulation results for the Failure Threads - Plot6

our proposed approach with the existing graph frameworks (Ligra and Galois). We also plan to apply our approaches for applications where iterative algorithms are the direction for future work.

References

[1] J. Gross and J. Yellen, *Graph Theory and Its Applications*. USA: CRC Press, Inc., 1999.

[2] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web." 1999.

[3] R. Motwani and P. Raghavan, "Randomized algorithms," in *Algorithms and Theory of Computation Handbook*, ser. Chapman & Hall/CRC Applied Algorithms and Data Structures series, M. J. Atallah, Ed. CRC Press, 1999. [Online]. Available: https://doi.org/10.1201/9781420049503-c16

[4] J. Vu, "The art of multiprocessor programming by maurice herlihy and nir shavit," *ACM SIGSOFT Softw. Eng. Notes*, vol. 36, no. 5, pp. 52–53, 2011. [Online]. Available: https://doi.org/10.1145/2020976.2021006

[5] S. Peri, C. K. Reddy, and M. Sa, "An efficient practical concurrent wait-free unbounded graph," in *21st IEEE International Conference on High Performance Computing and Communications; 17th IEEE International Conference on Smart City; 5th IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2019, Zhangjiajie, China, August 10-12, 2019*, Z. Xiao, L. T. Yang, P. Balaji, T. Li, K. Li, and A. Y. Zomaya, Eds. IEEE, 2019, pp. 2487–2494. [Online]. Available: https://doi.org/10.1109/HPCC/SmartCity/DSS.2019.00348

[6] B. W. Barrett, J. W. Berry, R. C. Murphy, and K. B. Wheeler, "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, May 23-29, 2009*. IEEE, 2009, pp. 1–8. [Online]. Available: https://doi.org/10.1109/IPDPS.2009.5161102

[7] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, A. Nicolau, X. Shen, S. P. Amarasinghe, and R. W. Vuduc, Eds. ACM, 2013, pp. 135–146. [Online]. Available: https://doi.org/10.1145/2442516.2442530

[8] P. Garg and K. Kothapalli, "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, January 4-7, 2016*. ACM, 2016, pp. 15:1–15:10. [Online]. Available: https://doi.org/10.1145/2833312.2833322

[9] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," in *UAI 2010, Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence, Catalina Island, CA, USA, July 8-11, 2010*, P. Grünwald and P. Spirtes, Eds. AUAI Press, 2010, pp. 340–349. [Online]. Available: https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=2126&proceeding_id=26

[10] I. Mitliagkas, M. Borokhovich, A. G. Dimakis, and C. Caramanis, "Frogwild! - fast pagerank approximations on graph engines," *CoRR*, vol. abs/1502.04281, 2015. [Online]. Available: http://arxiv.org/abs/1502.04281

[11] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy," in *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013. [Online]. Available: http://cidrdb.org/cidr2013/Papers/CIDR13_Paper58.pdf

[12] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, M. Kaminsky and M. Dahlin, Eds. ACM, 2013, pp. 456–471. [Online]. Available: https://doi.org/10.1145/2517349.2522739

[13] S. Beamer, K. Asanovic, and D. A. Patterson, "Reducing pagerank communication via propagation blocking," in *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. IEEE Computer Society, 2017, pp. 820–831. [Online]. Available: https://doi.org/10.1109/IPDPS.2017.112

[14] A. Panyala, O. Subasi, M. Halappanavar, A. Kalyanaraman, D. G. Chavarría-Miranda, and S. Krishnamoorthy, "Approximate computing techniques for iterative graph algorithms," in *24th IEEE International Conference on High Performance Computing, HiPC 2017, Jaipur, India, December 18-21, 2017*. IEEE Computer Society, 2017, pp. 23–32. [Online]. Available: https://doi.org/10.1109/HiPC.2017.00013

[15] Z. Peng, A. Powell, B. Wu, T. Bicer, and B. Ren, "Graphphi: efficient parallel graph processing on emerging throughput-oriented architectures," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT 2018, Limassol, Cyprus, November 01-04, 2018*, S. Evripidou, P. Stenström, and M. F. P. O'Boyle, Eds. ACM, 2018, pp. 9:1–9:14. [Online]. Available: https://doi.org/10.1145/3243176.3243205

[16] K. Lakhotia, R. Kannan, and V. K. Prasanna, "Accelerating pagerank using partition-centric processing," *CoRR*, vol. abs/1709.07122, 2017. [Online]. Available: http://arxiv.org/abs/1709.07122

[17] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[18] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: http://networkrepository.com

[19] L. Luo and Y. Liu, "Processing graphs with barrierless asynchronous parallel model on shared-memory systems," *Future Gener. Comput. Syst.*, vol. 106, pp. 641–652, 2020. [Online]. Available: https://doi.org/10.1016/j.future.2020.01.033

[20] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004*, M. W. Berry, U. Dayal, C. Kamath, and D. B. Skillicorn, Eds. SIAM, 2004, pp. 442–446. [Online]. Available: https://doi.org/10.1137/1.9781611972740.43