

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

Hemalatha Eedi¹ **Sathya Peri**¹ Neha Ranabothu¹ Rahul Utkoor¹

PDP 2021

¹Department of Computer Science & Engineering, IIT Hyderabad, India

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

1. Introduction
2. Related Work
3. Experimental Evaluation
4. Conclusion and Future Work

1. Introduction
2. Related Work
3. Experimental Evaluation
4. Conclusion and Future Work

Introduction: PageRank

- PageRank algorithm is a benchmark for many graph analytics

Introduction: PageRank

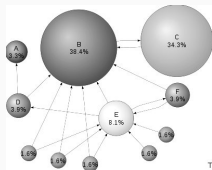
- PageRank algorithm is a benchmark for many graph analytics
- It is an iterative algorithm that updates ranks of pages until the value converges.

Introduction: PageRank

- PageRank algorithm is a benchmark for many graph analytics
- It is an iterative algorithm that updates ranks of pages until the value converges.

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

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.

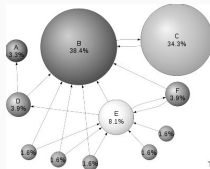


Introduction: PageRank

- PageRank algorithm is a benchmark for many graph analytics
- It is an iterative algorithm that updates ranks of pages until the value converges.

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

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.



- In each step, the algorithm approximates the order of page

Introduction: System Model

- 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 primitive - CAS(Compare-And-Swap).

Listing 1: CAS function

```
1 CAS(int expected, int updated) {  
2   int prior = this.value  
3   if(this.value == expected) {  
4     this.value = update;  
5     return true;  
6   }  
7   return false;  
8 }
```


- Blocking Synchronization
 - Uses locks to allow one thread at a time to access a shared object
 - Prevents conflicts between the coordinating threads
 - However, it results in busy waiting and deadlocks conditions
- Non-Blocking Synchronization
 - 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

1. Introduction
2. Related Work
3. Experimental Evaluation
4. Conclusion and Future Work

Parallel computation of the PageRank metric on graphs has been studied extensively on shared memory architectures using many different programming models in recent years.

Solution	PageRank Approach	Barriers	Conclusions Drawn
Garg et al. ¹	STICD	Yes	Redundant computations are removed and , the preprocessing techniques used in this work are not parallelized
Beamer et al. ²	Propagation Blocking	Yes	Reduced Memory Bound Computations and Improves Spatial Locality
Ajay Panyala et al. ³	loop perforation	Yes	Imprived performance and uses extra memory
Zhen Peng et al. ⁴	GraphPhi	Yes	Benifited with data-locality, effective scheduling, and load balancing

¹P. Garg, K. Kothapall: STIC-D: algorithmic techniques forefficient parallel pagerank computation on real-world graphs. ICDCN, 2016.

²S. Beamer, K. Asanovi, D. A. Patters : "Reducingpagerank communication via propagation blocking. IPDPS, 2017

³A. Panyala, O. Subasi, M. Halappanavar, A. Kalyanaraman, D. G.Chavarr ia-Miranda, S. Krishnamoorthy:Approximate computin gtechniques for iterative graph algorithms. HiPC, 2017.

⁴Z. Peng, A. Powell, B. Wu, T. Bicer, B. Re: "Graphphi:efficient parallel graph processing on emerging throughput-oriented architectures. PACT, 2018.

Description of Algorithms

- Most of the research on PageRank computation is on Graph pre-processing step
- Most of 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

Baseline Barrier Synchronization Algorithm

- In each iteration every thread is allocated with equal amount of work
- Threads after computing the PageRank of their allocated vertices has to wait for other threads at the end of the iteration
- Pseudo Code

```
1: for all  $u \in \text{threadVertices}(T_i)$  do
2:    $pr(u) \leftarrow \frac{(1-d)}{n}$ 
3:   for all  $u \in V$  such that  $(v,u) \in E$  do
4:      $pr(u) = pr(u) + \frac{prPrev(v)}{outDeg(v)} * d$ 
5:   end for
6:    $thrErr[T_i] = \max(thrErr[T_i], |prPrev(u) - pr(u)|)$ 
7: end for
8:
9: Barrier checkpoint
10: for all threads  $T_i \mid i \in \{1, \dots, p\}$  do
11:    $error = \max(error, thrErr[T_i])$ 
12: end for
```

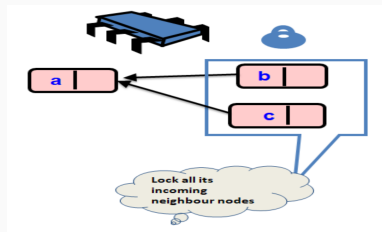
Fine-Grain Lock Variant 1

Threads are allowed to compute at any iteration without Barrier

- Read-Write conflicts are handled by using locks
- The vertices in each partition are categorized into internal and boundary vertices.
- Thread acquires the locks on all incoming vertices to compute PR for boundary vertices
- Thread error is updated locally and a global lock is used at the end of each iteration to update the Thread error to max of all thread errors

Fine-Grain Lock Variant 1

```
1: for all  $u \in \text{internalVertices}(T_i)$  do
2:    $pr(u) \leftarrow \text{ComputePR}(u)$ 
3: end for
4: for all  $u \in \text{boundaryVertices}(T_i)$  do
5:   for all  $v \in V$  such that  $(v, u) \in E \cup u$  do
6:      $v.\text{lock}()$ 
7:   end for
8:    $pr(u) \leftarrow \text{ComputePR}(u)$ 
9:   for all  $v \in V$  such that  $(v, u) \in E \cup u$  do
10:     $v.\text{unlock}()$ 
11:   end for
12: end for
```



Fine-Grain Lock Variant 2

Threads are allowed to compute any iteration without Barrier

- Instead of locking all incoming nodes at once
- Lock each incoming node
- Read its value (add to $pr(u)$) and release the lock
- Repeat this process for all incoming nodes

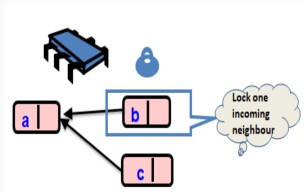


Figure 1: Internal Representation

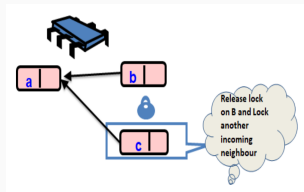


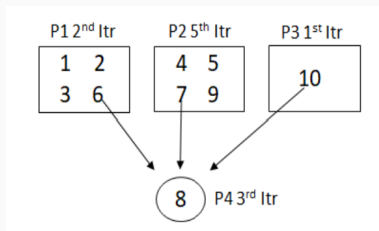
Figure 2: Internal Representation

Algorithm 1 Lock variant 2

```
1: for all  $v \in V$  such that  $(v, u) \in E$  do  
2:    $v.\text{lock}()$   
3:    $temp = temp + \frac{pr(v)}{outDeg(v)} * d$   
4:    $v.\text{unlock}()$   
5: end for
```

Non-Blocking Algorithms

- No-Synch Algorithm eliminates locks and computes PR values using atomic operations.
- Approximate version of Barrier



Non-Blocking Algorithms

```
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
```

```
1: for all  $u \in threadVertices(T_i)$  do
2:    $prev \leftarrow pr(u)$ 
3:    $temp \leftarrow ComputePR(u)$ 
4:    $pr(u).store(temp)$ 
5:    $thrlocErr = \max(thrlocErr, |temp - prev|)$ 
6: end for
7:  $thErr[T_i].store(thrlocErr)$ 
8:  $localError \leftarrow 0$ 
9: for all  $tid \in threads(1, p)$  do
10:
11:    $localError = \max(localError, thErr[tid].load())$ 
11: end for
```

Lemma-1

The algorithm eventually terminates in finite steps

- As a base case, threads can be considered to be present in two consecutive iterations at a particular instant
- According to Base Algorithm Equation 2 is

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

$$err_i^u = |pr_i^u - pr_{i-1}^u| \quad (3)$$

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

$$|pr_{i-1:i}^u - pr_{i-1}^u| \leq |pr_i^u - pr_{i-1}^u| \Rightarrow err_{i-1:i}^u \leq err_i^u \quad (4)$$

$$pr_{i-1:i}^u = \frac{1-d}{n} + d * \sum_{v \in S_i^u} \frac{pr_{i-1}^v}{outDeg(v)} + d * \sum_{v \in S_{i-1}^u} \frac{pr_{i-2}^v}{outDeg(v)} \quad (5)$$

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

$$err_{i-1:i}^u = |pr_{i-1:i}^u - pr_{i-1}^u| \quad (6)$$

Lemma-2

The algorithm leads to a similar result as that of Sequential

- 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

$$\widehat{pr}_i^u = \frac{1-d}{n} + d * \sum_{l=1}^l \sum_{v \in S_l^u} \frac{\widehat{pr}_l^v}{outDeg(v)} \quad (7)$$

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

- The PageRank values from the algorithm are also similar to that of the Sequential output with an error which is less than the threshold
 $|pr^u - \widehat{pr}^u| \leq threshold$

Wait-Free Algorithm

- Ensures algorithm correctness. Gives exact PR values as that of base algorithm
- Threads are not allowed to enter into the next iteration until all nodes are computed
- 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

Non-Blocking Algorithms-Wait-Free Algorithm

```
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;  
};
```

- Thread Object to store current iteration, current node till which PR computation is done and thread error until current node. Useful for helper thread to continue the computation of left over nodes for the partition
- Global Object to store current iteration (incremented only if all nodes are computed), error from all threads for itr. All threads update this global object with their max error
- Node Object to store the PR value and iteration number. Itr is incremented after updating the rank. Useful to know for helper threads whether to compute the PR for the node

Non-Blocking Algorithms-Wait-Free Algorithm

```
1: procedure UPDATEPR(u, nodePr, thdVar)
2:   z ← pr(u)
3:   if z.itr == thdVar.itr then
4:     casOb ← newPrCasOb(thdVar.itr + +, nodePr)
5:     CAS(pr(u), z, casOb)
6:   end if
7:   z ← prevPr(u)
8:   if z.itr == thdVar.itr then
9:     casOb ← newPrCasOb(thdVar.itr + +, nodePr)
10:    CAS(prevPr[u], z, casOb)
11:   end if
12: end procedure
```

```
1: z = glbThInfo[hlpld]
2: if z.itr == thdVar.itr then
3:   er ← max(z.er, |nodePr - prevPr|)
4:   casOb ← newThCASOb(z.itr, next(u, hepld), er)
5:   CAS(glbThInfo[hlpld], z, casOb)
6: end if
```

```
1: procedure UPDATEGLBVAR((thld, hlpld, thdVar)
2:   while true do
3:     z ← glbVar
4:     if z.itr == thdVar.itr then
5:       casOb ← copy(z)
6:       casOb.check[hlpld] ← true
7:     casOb.er ← max(casOb.er, glbThInfo[hlpld].er)
8:     CAS(glbVar, z, casOb)break
9:   end if
10:  end while
11:  while true do
12:    z ← glbThInfo[hlpld]
13:    if z.itr == thdVar.itr then
14:      casOb ← newThCASOb(z.itr + +, thld, 0)
15:      CAS(glbThInfo[hlpld], z, casOb)break
16:    end if
17:  end while
18: end procedure
```


1. Introduction
2. Related Work
3. Experimental Evaluation
4. Conclusion and Future Work

- Platform.
 1. Intel(R) Xeon(R) E5-2660 v4 processor architecture, 2.06 GHz core frequency, 56 cores, 32GB RAM
 2. Compiler - g++ 7.5.0 with POSIX MultiThreaded library support
- Datasets:
 1. Synthetic datasets(#vertices: $2^{21} \sim 2^{23}$, $1 \sim 7 \times 10^6$)
 2. Real-world datasets from snap(vertices: $1 \sim 7 \times 10^6$)

Results: PageRank Speed-Up w.r.t Identical Nodes

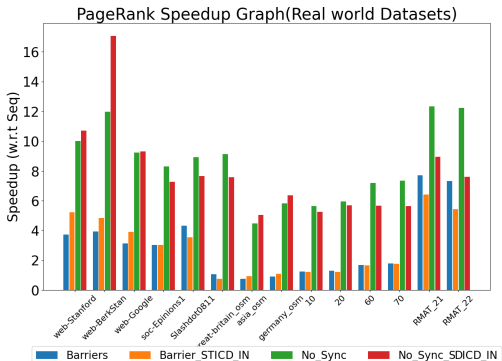


Figure 3: Speed-Up on Real-World, Synthetic Datasets

- No_Sync provide an average speed up of 5.1x over Barrier
- Our proposed approach on Web-graphs, Social-networks, Road-networks, Synthetic datasets follows similar pattern when incorporated with *Identical nodes* optimization from STICD

Results: PageRank Speed-Up w.r.t chain of nodes

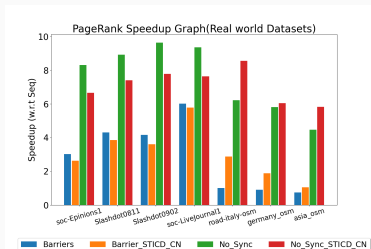


Figure 4: Real-World Datasets

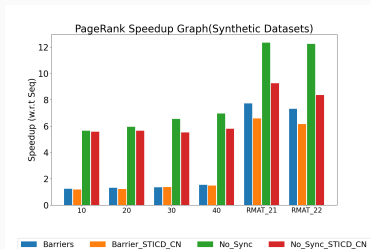


Figure 5: Synthetic Datasets

- No_Sync provide an average speed up of 4.3x over Barrier
- Our proposed approach on Social-networks, Road-networks, Synthetic datasets follows similar pattern when incorporated with *Chain nodes* optimization from STICD

Results: PageRank Time with Random Thread Sleep

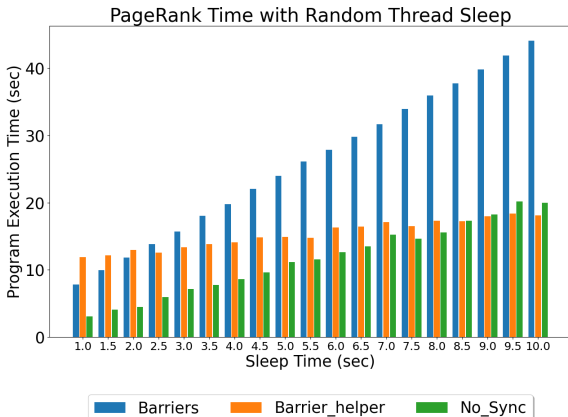


Figure 6: PageRank Time with Random Thread Sleep

- Deterministically added sleeps to the threads in selected iteration
- Execution time of Barrier and No_Sync variants increases with increase in sleep time, whereas wait-free execution time is consistent

Results: PageRank with Thread Failures

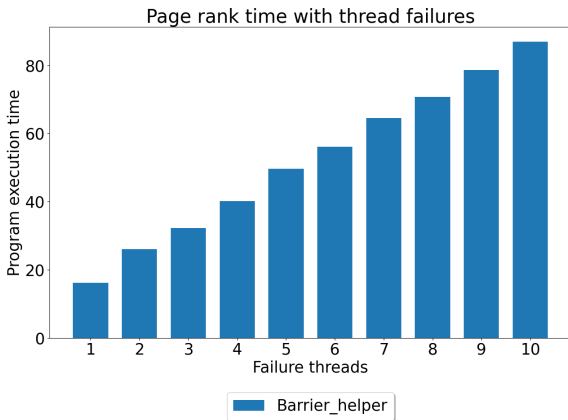


Figure 7: PageRank with Thread Failures




- Barrier_helper parallel variant handle thread failures
- Other variants fail to handle this property
- Increase in number of thread failures, increases program execution

1. Introduction
2. Related Work
3. Experimental Evaluation
4. Conclusion and Future Work



- We developed barrier-less implementations of PageRank algorithm
- On average our No_Sync variant is 4x times faster than barrier
- We developed Wait-free(Barrier_helper) variant of PageRank algorithm, which handles thread-failures

- Currently we incorporated techniques from STICD, adding more such optimization techniques is our primary goal
- Our wait-free(Barrier_helper) performance is poor compared to barrier. Improving wait-free algorithm is our secondary goal

References

-  [1] Page et al. “The PageRank Citation Ranking: Bringing Order to the Web.” Accessed: November, 1999.
<http://ilpubs.stanford.edu:8090/422/>
-  [2] Garg et al. “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, pp. 15:1–15:10. ACM, 2016. <https://doi.org/10.1145/2833312.2833322>
-  [3] Beamer et al. Reducing pagerank communication via propagation blocking,” in 2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, pp. 820–831. IEEE, 2017.
<https://ieeexplore.ieee.org/document/7967173>

References

-  [4] A. Panyala et al., “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). Springer, Cham.
<https://ieeexplore.ieee.org/document/8287732>
-  [5] Z. Peng et al., “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. Evrpidou, P. Stenstrom, and M. F. P. O’Boyle, Eds. ACM, 2018, pp. 9:1–9:14.
<https://dl.acm.org/doi/10.1145/3243176.3243205>

Thank You