PageRank Algorithm
Serial and Parallel Implementation

Nidhi Manek
HPC, DAIICT
Gandhinagar, India
201901402@daiict.ac.in

Krupal Patel
HPC, DAIICT
Gandhinagar, India
201901406@daiict.ac.in

Vidhi Badrakiya
HPC, DAIICT
Gandhinagar, India
201901434@daiict.ac.in

Vishv Joshi
HPC, DAIICT
Gandhinagar, India
201901453@daiict.ac.in

Ashray Kothari
HPC, DAIICT
Gandhinagar, India
201901457@daiict.ac.in

Kavya Parikh
HPC, DAIICT
Gandhinagar, India
201901474@daiict.ac.in

***Abstract*—Pagerank is an algorithm to rank the web pages in search engine results according to their importance and behavior. In this paper, we investigate how to effectively parallelize the pagerank algorithm using high-performance computing principles, with the goal of improving scalability with increased compute power and improved productivity through increased usage of existing compute resources.**

I. Introduction

In the era of internet where plethora of information is available on a simple click, the challenge is to get the best possible search results. PageRank algorithm was developed for the very same purpose. The purpose of this algorithm is to rank the results of a particular search. PageRank algorithm forms an essential core of many search engines like Google and Yahoo. It has evolved to an extent that it is now used to rank the users on social media in order to provide better services and entertainment. As described by Google, PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.

A. History

PageRank was first introduced by the founders of Google Sergey Brin and Larry Page. Larry Page and Sergey Brin developed PageRank at Stanford University in 1996 as part of a research project about a new kind of search engine. Sergey Brin had the idea that information on the web could be ordered in a hierarchy by "link popularity": a page ranks higher as there are more links to it. The system was developed with the help of Scott Hassan and Alan Steremberg, both of whom were cited by Larry Page and Sergey Brin as being critical to the development of Google. Rajeev Motwani and Terry Winograd co-authored with Page and Brin the first paper about the project, describing PageRank and the initial prototype of the Google search engine, published in 1998. The name "PageRank" plays on the name of developer Larry Page, as well as of the concept of a web page. Owing to its origins, this algorithm is also known as Google algorithm. [1]

B. Literature Review

The PageRank algorithm was first introduced in the paper by Sergery Brin and Larry Page titled *The PageRank Citation Ranking: Bringing Order to the Web* [2]. This paper gives an insight into the need to develop this algorithm. It highlights the fact that if the ranking was based on citations alone, a simple program could easily manipulate the results by increasing the number of citations. This also means that any replicable features of a web page should not be used as a metric to evaluate the rank. Furthermore, a web page is composed of a wide variety of contents and structure. In other words, it is non-standardized and unregulated. So, the paper mentions an approximation that a webpage's importance can be determined by how many webpages it has been linked to in the net. As it was the first paper which introduced this algorithm, the related mathematics has been described elaborately.

The paper titled *An Improved Page Rank Algorithm based on Optimized Normalization Technique* [3] shows an improvement to the original technique by introducing normalization. The normalization is achieved by calculating the mean value of page ranks and then dividing the value of page rank by the mean value of page ranks. The proposed scheme reduces the time complexity of the traditional Page Rank algorithm by reducing the number of iterations to reach a convergence point.

The paper titled *Weighted PageRank Algorithm* [9] gives an analysis of putting weights on the links between webpages. This paper shows relevancy calculations which indicate that the performance or relevance is either equal or better for different values of page set size for the Weighted PageRank algorithm.

The paper titled *Site-Based Partitioning and Repartitioning Techniques for Parallel PageRank Computation* [8] discusses the results of using state-of-the-art sparse matrix partitioning

models in order to attain high efficiency in parallel Page Rank computations with a particular focus on reducing the preprocessing overhead. For this purpose, it evaluates two different compression schemes on the web matrix using the site information inherently available in links. The paper also discusses on extending this technique to initially distributed data due to memory limitations in a single system. The distrubuted data is re-partitioned for efficient Page Rank computation.

The paper titled *Distributed pagerank for P2P systems* [7] defines and describes a fully distributed implementation of Google's highly effective pagerank algorithm, for "peer to peer" (P2P) systems. The implementation is based on chaotic (asynchronous) iterative solution of linear systems. The P2P implementation also enables incremental computation of pageranks as new documents are entered into or deleted from the network. Incremental update enables continuously accurate pagerank. The paper also establishes that the computation is extremely fast for an incremental update due to rapid convergence with very accurate results.

The paper titled *Distributed PageRank computation based on iterative aggregation-disaggregation methods* [4] examines a solution to compute pagerank of a distributed system while optimizing communication cost which arises from the fact that the traditional pagerank algorithm needs the global link graph information to provide the necessary results. This paper proposes a distributed PageRank computation algorithm based on iterative aggregation-disaggregation (IAD) method with Block Jacobi smoothing. The basic idea is divide-and-conquer. Each website is treated as a node to explore the block structure of hyperlinks. Local PageRank is computed by each node itself and then updated with a low communication cost with a coordinator.

The paper titled *Efficient PageRank and SpMV Computation on AMD GPUs* [5] focuses on making the PageRank algorithm run faster on AMD GPUs. The core of the algorithm is Sparse Matrix Vector Multiplication. The paper introduces a faster method to run these computations by using a Compressed Sparse Row format. The linkage matrices used in PageRank are always very sparse, with highly uneven row sizes (number of non-zero values in a row of a matrix). Based on the row sizes, the rows are sorted into three bins, and assigned different number of threads to process the rows in different bins. This improves load balance and hardware resource utilization. This routine is implemented in AMD GPU architecture.

The paper titled *Deriving Highly Efficient Implementations of Parallel PageRank* [15] tries to speed up the calculation speed of the original PageRank paper [2] by using forelem framework.

The paper titled *An Efficient Practical Non-Blocking PageRank Algorithm for Large Scale Graphs* [16] presents algorithms to compute pagerank on a shared memory systems. The paper presents parallel implementations of PageRank algorithm using barrer and lock variants. The paper then goes on to show barrier-less synchronization and lock-free implemenatations of parallel PageRank to overcome the issues of lock based methods. 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. As a result, to achieve non-blocking progress, the threads executing cannot acquire locks since a thread that has acquired a lock can potentially block other threads from progressing by becoming slow or crashing.

C. Project Objectives in terms of parallelization:

The large and heterogeneous nature of the web have created many challenges for information retrieval. Despite a rich source of existing works, the algorithms proposed previously are very slow for real-time search. Additionally, Page rank computation should also tackle issues like rank leakage, rank sinks and optimal convergence. Extensive research has been conducted over the past several years to improve and optimize the page-rank algorithm. Parallel pagerank computation was proposed to achieve a faster convergence rate and minimize convergence rate, the principle being to utilize the strength of large cluster based computation. Zhu Y, Ye S, Li X [1] proposed a distributed page rank calculation method that made use of Iterative Aggregation-Disaggregation (IAD) method, which in turn made use of the divide and conquer technique, to speed up page rank computation in a shared memory space. Then, Wu T, Wang B and Shan Y's power method [2], which works recursively till a threshold is reached, further improved the computational time 3-4 times. The scalability of pagerank on the large scale supercomputer TSUBAME was evaluated by Cevahir A. Aykanat C [3]. Further, techniques like monte carlo method, fully distributed peer-to-peer architecture [5], partitioning and repartitioning techniques for computation [6], were proposed which enhanced the existing algorithms. In further project, we will work on the accelerated parallel page rank algorithm (APPR) and will compare its efficiency over the serial algorithm.

D. Contributions

In this paper we implement and parallelize the Pagerank algorithm using OpenMP directives in shared memory architecture in C language. The key contributions are as follow:

- We explain the serial implementation of pagerank algorithm.
- We parallelize the serial code using different openMP directives and scheduling mechanisms.
- We compare and analyze our Pagerank implementations with state-of-the art algorithm for two different datasets.
- We present the profiling and load-balance analysis of the parallel implementation of the pagerank algorithm.
- We have attempted to analyse the time complexity of the parallel state of the art implementation and compared it with our parallel implementation.

II. Serial Implementation

A. Serial Algorithm

The Pagerank algorithm was originally developed to rank web pages. For the algorithm, the web network is represented

as a directed graph G=(V,E) network where web pages are vertices and links between web pages are edges.

The pagerank of a web page is calculated using the pageranks of web pages which are pointing to it. Iterative formula to compute pagerank of web page P_i can be given as:

$$PR_{k+1}(P_i) = \sum_{P_j} \frac{PR_k(P_j)}{C(P_j)} \tag{1}$$

where, $PR(P_i)$ = pagerank of vertex P_i
k = no. of iteration
P_j = Vertices P_j which are pointing vertex P_i
C(P) = no. of outgoing links from vertex P

Formula shows, Pagerank of a web page will be divided into the web pages it is pointing. Which states the web page which has more links coming from other pages is more famous, having higher pagerank value.

The pagerank algorithm described above works absolutely well but sometimes problems happen in this approach when we have Dangling Vertices in the network. Dangling Vertices are vertices which have no outgoing links. Sometimes network graph may have some disconnected independent components, then in this case also the algorithm mention above will not work properly.

This problem lead us to introduce teleportation factor d and to define the final formula for the pagerank algorithm. If not given, d is usually taken as 0.85. The final formula of Pagerank algorithm is as follow:

$$PR_{k+1}(P_i) = \frac{(1-d)}{n} + d * (\sum\nolimits_{P_j} \frac{PR_k(P_j)}{C(P_j)}) \tag{2}$$

This graph mathematics can easily be done using matrix operations. We can represent graph as adjacency matrix based on graph parameters. adjacency_matrix[i][j] is 1 if there is outgoing link from i^{th} vertex to j^{th} vertex otherwise it is 0. Initial pageranks of all vertices(PR matrix) will be 1/n where n is number of vertices in the network, which shows initially all the web pages are equally ranked using equal probability distribution. Now, We can define matrix H which will be used to compute PR matrix for next iteration. H can be computed using num_of_outlinks matrix which contains number of outgoing links for each vertex. If adjacency_matrix[i][j]=1 then assign H[i][j] as $\frac{1}{num\_of\_outlinks[i]}$ otherwise 0. PR matrix for next iteration can be obtained by multiplying previous PR matrix and H matrix.

$$PR_{k+1} = \frac{(1-d)}{n} + d * (PR_k * H) \tag{3}$$

After a number of iterations pagerank value for each web page will converge to constant value as shown in Fig. (1) and we can rank web pages according to their pagerank value. We can state a web page with higher pagerank value is more important.

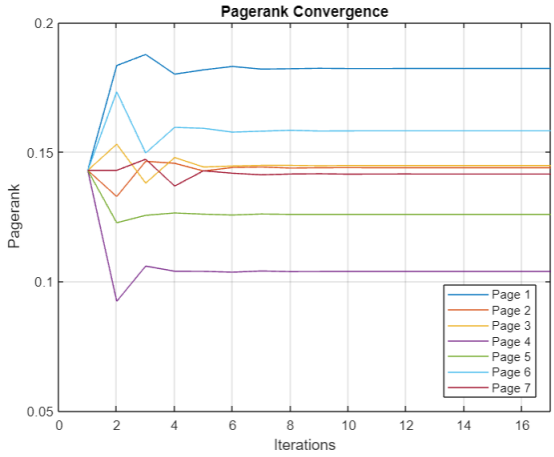

Fig. 1: PageRank Convergence

B. Complexity

If V is no. of vertices in the network:
Time-complexity of the algorithm = O(V^2)
Auxiliary space-complexity = O(V)
Total space-complexity = O(V^2)

C. Example

Let us see an example using the following graph.

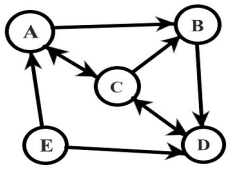

Fig. 2: Example

There are links from,

A → B, C
B → D
C → A, B, D
D → C
E → A,D

Using this we calculate the Transition Matrix. From node A there are two possibilities to go to, B and C. Thus, their respective probabilities are 1/2 each (1/number_of_outlinks). Similarly we calculate for the rest of the nodes.

Afterwards we calculate PR by multiplying the previous PR with the Transition Matrix using eq. (3). We set the initial value of the pagerank for each node to be equal (1/Total Nodes).

Algorithm 1 Procedure Pagerank (Adj, n, tolerance)

PR ← 0 /* *Declare a matrix PR of size 1xn to store pagerank of all vertices and assign 1/n to all the matrix elements* */

PR_prev ← 1/n /* *Define PR_prev matrix to store the previous iterations of PR matrix* */

num_outlinks ← 0 /* *Declare a matrix num_outlinks of size 1xn to store number of outgoing links of all vertices and assign 0 to each index* */

for i = 1 to n **do**
  for j = 1 to n **do**
    if Adj[i][j]==1 **then**
      num_outlinks[i] ← num_outlinks[i] + 1
    end if
  end for
end for

H ← 0 /* *Declare a 2D matrix H of size nxn to get the PR of next iteration* */
for i = 1 to n **do**
  for j = 1 to n **do**
    if Adj[i][j]==1 **then**
      H[i][j] ← $\frac{1}{num\_outlinks[i]}$
    end if
  end for
end for

while (maximum(abs(PR-PR_prev))>tolerance) **do**
  PR_prev ← PR
  PR ← 0
  for i=1 to n **do**
    for j=1 to n **do**
      PR[i] ← PR[i]+PR_prev[i]∗H[j][i]∗d
    end for
    PR[i]← PR[i]+ $\frac{1-d}{n}$
  end for
end while

return PR

	A	B	C	D	E
A	0	0	1/3	0	1/2
B	1/2	0	1/3	0	0
C	1/2	0	0	1	0
D	0	1	1/3	0	1/2
E	0	0	0	0	0

TABLE I: Transition Matrix

Thus, webpage C has the highest pagerank and it is most important of all and webpage E has lowest pagerank and so, it is least important.

	A	B	C	D	E
Itr0	0.200	0.200	0.200	0.200	0.200
Itr1	0.172	0.172	0.285	0.342	0.030
Itr2	0.123	0.184	0.393	0.269	0.030
Itr3	0.154	0.194	0.311	0.310	0.030
Itr4	0.131	0.184	0.359	0.296	0.030
Page Rank	2	3	5	4	1

TABLE II: Page Rank Calculation

D. Flowchart of serial algorithm

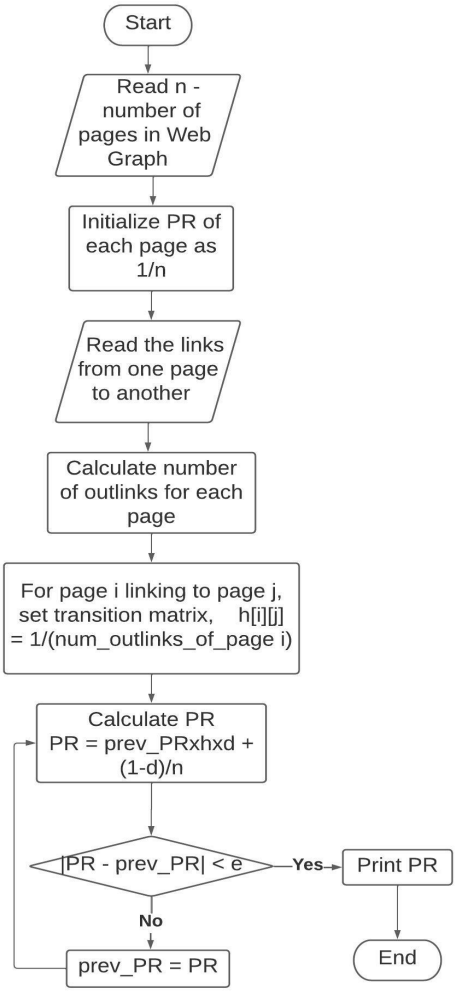

Fig. 3: Serial Algorithm Flowchart

E. Dataset

Dataset Name: Gnutella peer-to-peer network
Date (Dataset 1): August 25 2002
Date (Dataset 2): August 24 2002
Gnutella is a file sharing protocol that defines the way distributed nodes communicate over a peer-to-peer (P2P) network. In August 2002, nine snapshots of the Gnutella network were taken. In the Gnutella network structure, nodes represent hosts and edges represent connections between Gnutella hosts. [10]

The statistics for both the datasets can be seen in table (III) and (IV).

Dataset Statistics - p2p-Gnutella25	
Nodes	22,687
Edges	54,705
Nodes in Largest WCC	22,663
Edges in Largest WCC	54,693
Nodes in Largest SCC	5153
Edges in Largest SCC	17695

TABLE III: Dataset 1 Statistics

Dataset Statistics - p2p-Gnutella24	
Nodes	26518
Edges	65369
Nodes in Largest WCC	26498
Edges in Largest WCC	65359
Nodes in Largest SCC	6352
Edges in Largest SCC	22928

TABLE IV: Dataset 2 Statistics

F. Why Computationally Intensive

The pagerank algorithm described above has a time complexity of $O(V^2)$ where V is the number of nodes. As of now billions of web pages are available on internet and also as number of web pages are increasing at every second and also they are getting linked with different web pages. If we try to follow the described algorithm, it would take large amount of time to search anything on the internet. It is very difficult to access, process and rank them and also respond to user when such a huge data is available. In web network it is difficult to compute the pagerank of even one web page and doing this for billions of web pages in real life would take days or even such months to compute it. Which shows it is highly computationally intensive and we can't work on this on a single machine. There should be optimized techniques and numerous machines working parallel on the algorithm.

G. Goals and deliverables

Our goal is to implement a parallel computation scheme which in turn will increase the speed at which results are calculated. The reduced time for computation will deliver a better user experience when searching for web pages. We will compare the difference of execution time between the serial code and the parallel computation code which will tell us exactly how effective is the parallel implementation scheme. The speed of getting the results and the relevance of the search results will determine the practicality of the applied parallelization techniques.

H. Memory bound

Our pagerank algorithm is memory bound problem because when we increase our problem sizes our data structure (vector in program point of view) will also increase and accessing the data from memory is a limitation because we can't store our data in cache all the times so there will be more cache misses.

III. Parallel Implementation

A. Implementation Details

Different part of serial code is parallelized using openMP directives. The for loop to calculate number of outlinks for every nodes is parallelized using for and Reduction clause. The nested for loop to compute H matrix is parallelized using for and collapse clause. The inner for loop to calculate PR matrix using previous PR matrix is parallelized using for and reduction clauses. Other implementation details remain same as the serial implementation.

B. Parallel Algorithm

C. Parallel time complexity

Time complexity of parallel implementation is $O(V^2/p)$ where p is number of threads as We parallelize each for loop in our code.

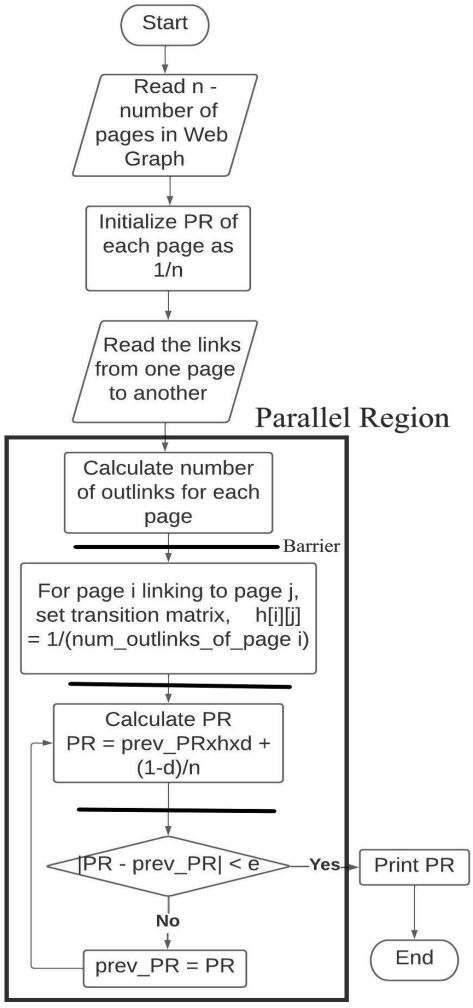

Fig. 4: Parallel Algorithm Flowchart

D. Correctness of Parallel Implementation

We ran the code of our verified serial implementation and used the result as true values for comparison to establish the

Algorithm 2 Procedure Parallel Pagerank (Adj, n, tolerance)

PR ← 0 /* *Declare a matrix PR of size 1xn to store pagerank of all vertices and assign 1/n to all the matrix elements* */

PR_prev ← 1/n /* *Define PR_prev matrix to store the previous iterations of PR matrix* */

num_outlinks ← 0 /* *Declare a matrix num_outlinks of size 1xn to store number of outgoing links of all vertices and assign 0 to each index* */

for i = 1 to n **do**
  num_outlinks1 ← 0 // Global variable to store reduction result

  #pragma omp parallel for reduction(+:num_outlinks1) <schedule>
  for j = 1 to n **do**
    if Adj[i][j]==1 **then**
      num_outlinks1 ← num_outlinks1 + 1
    end if
  end for
  num_outlinks[i] ← num_outlinks[i]+num_outlinks1

end for

H ← 0 /* *Declare a 2D matrix H of size nxn to get the PR of next iteration* */
#pragma omp parallel for <schedule> collapse(2)
for i = 1 to n **do**
  for j = 1 to n **do**
    if Adj[i][j]==1 **then**
      H[i][j] ← $\frac{1}{num\_outlinks[i]}$
    end if
  end for
end for

while (maximum(abs(PR-PR_prev))>tolerance) **do**
  PR_prev ← PR
  PR ← 0

  for i=1 to n **do**
    PR1 ← 0 // Global variable to store reduction result
#pragma omp parallel for reduction(+:PR1) <schedule>
    for j=1 to n **do**
      PR1 ← PR1+PR_prev[i]∗H[j][i]∗d
    end for
    PR[i]← PR1+ $\frac{1-d}{n}$
  end for
end while

return PR

validity of our parallel implementation. We ran our parallel implementation code for various numbers of threads. The results for both serial and parallel implementation for various processors came out to be the same, indicating that our results are consistent.

E. State of the Art Implementation

The implementation that we used as a benchmark to compare our implementation uses the map data structure instead of matrix as in our implementation. The map data structure significantly reduces the space and time complexity. Our parallel implementation uses $O(V^2)$ space complexity where V represents number of vertices in the dataset. The state of the art implementation's use of map data structure will create an adjacency list to store the graph. The space complexity here will be $O(V + E)$. Thus, we see that the space complexity for most of the problems will be significantly lesser than our implementation. The only way the state of the art algorithm will use more memory than our implementation is when the graph will have too many edges compared to the number of vertices.

With regards to time complexity, our implementation is $O(V^2/p)$ whereas for state of the art it is $O(\frac{V}{p} + E_i)$ where E_i is the maximum of total number of edges in the p groups divided and i is the corresponding thread number [13]. Similar argument can be made regarding the time complexity that our implementation will rival the state of the art algorithm's performance only when the number of edges are too large for a given number of vertices. In our implementation, to find all neighbours of any node the program needs to traverse complete row of adjacency matrix due to which time complexity ($O(V)$) is more (especially when we have sparse matrix i.e. matrix with less entry). But to do the same thing in adjacency list(state of the art) is easy and less time consuming(basically O(degree of the vertex))

The main drawback of the state of the art implementation is that the memory usage increases if we increase the number of edges for the same number of vertices. This is not the case for our implementation in which the space complexity is determined by the number of vertices and is independent of the number of edges. This means that unless more webpages are added, our implementation uses the same amount of memory regardless of whether the number of connections between the webpages increase or decrease. This may make memory allocation more manageable as we do not have to assign new memory every time a webpage is linked to another webpage.

IV. Results & Profiling

	Runtime	Throughput	Speedup	Efficiency
Serial	37	1.8657	-	-
Parallel(2 threads)	19	3.6331	1.9000	0.9736
Parallel(3 threads)	12	5.7535	2.7142	1.0270
Parallel(4 threads)	9	7.6700	3.4545	1.0270
State of the art(4 threads)	0.84	82.1789	44.0476	11.0119

TABLE V: State of the art comparison with implemented serial and parallel algorithms for Dataset 1

	Runtime	Throughput	Speedup	Efficiency
Serial	38	2.4819	-	-
Parallel(2 threads)	20	4.7155	1.9473	0.9500
Parallel(3 threads)	14	6.7366	3.0830	0.9047
Parallel(4 threads)	11	8.5737	4.1110	0.8636
State of the art(4 threads)	0.86	109.6643	43.0232	10.7558

TABLE VI: State of the art comparison with our own implemented serial and parallel algorithms for Dataset 2

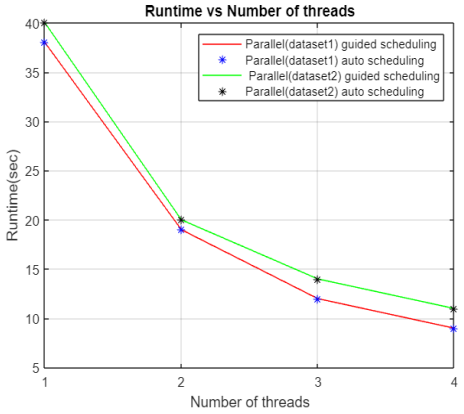

Fig. 5: Runtime vs number of threads

Runtime: As seen from the graph, the runtime for serial code is maximum. Due to parallelizing the code, the runtime decreases as number of threads increase. Thus, for 2 threads runtime is maximum and for 4 threads it is minimum as the work gets divided. For state of the art, the runtime is minimum.

Speedup: The speedup of parallel is higher as compared to serial code. After this, the phenomenon of parallel slowdown is achieved. It occurs in parallel computing where parallellization of an algorithm beyond a certain point causes it to run slower. The speedup is maximum for state of the art.

Efficiency: The efficiency decreases as we parallelize the code. This happens because more time is taken to parallelize the code. So, as the number of threads increase, time taken to synchronize increases and thus the efficiency decreases.

Throughput: The throughput increases when we parallelize the code as runtime decreases. As the number of threads increase, the throughput increases.

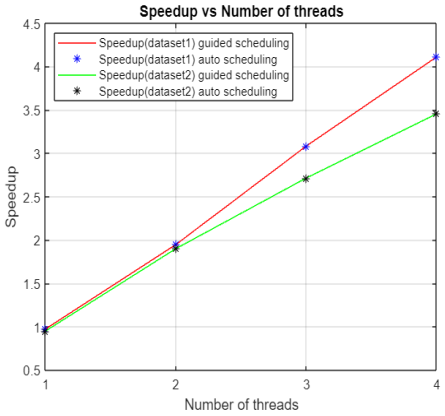

Fig. 6: Speedup vs number of threads

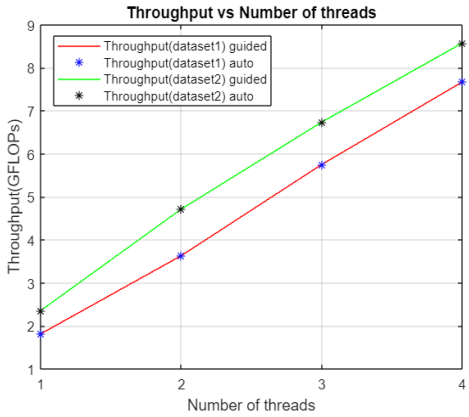

Fig. 7: Throughput vs number of threads

A. Load balancing

We have done analysis of load balance for p2p-Gnutella24 dataset (dataset 2). OpenMP provides different scheduling mechanisms to divide work among different threads. Here, load balance analysis is done for guided and auto scheduling mechanisms and also for without scheduling using 4 threads. Results are as shown in the figure(8).

As shown in the figure(8), time taken by different threads is close to each other for auto and guided mechanisms while in without scheduling method, time taken by each thread is varying, which shows some of the threads will have to wait idly when other threads are working and this will also increase overall run-time of the algorithm. In guided and auto scheduling mechanisms, Compiler will distribute work among the threads at run-time depending on which thread is free. So, threads will get used optimally, which will also decrease overall run-time of the algorithm.

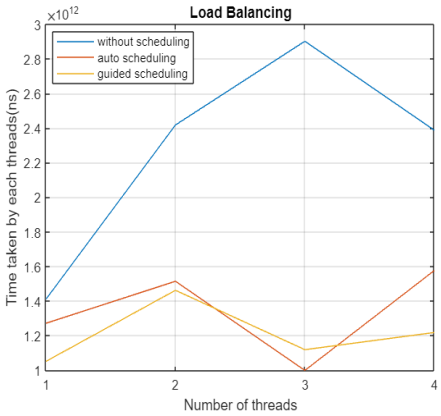

Fig. 8: number of threads vs time taken by each thread

We have done profiling using gprof, vtune and perf command for p2p-Gnutella24 dataset on Lab207 PC.

Lab207	
Architecture	x86_64
CPU(s)	4
Thread(s) per Core	1
Core(s) per Socket	4
Socket(s)	1
Clock Speed	3.3 GHz
L1d Cache	32KB
L1i Cache	32KB
L2 Cache	256KB
L3 Cache	6144KB

TABLE VII: Benchmark of the Lab207

Metrics	Serial	Parallel	SotA
Memory Bound(Pipeline Slots)	80.1%	73.7%	59.3%
Cache Bound(Clockticks)	71.4%	58.7%	47.9%
DRAM Bound(Clockticks)	21.2%	27.5%	0.0%
LLC Miss Counts	115,503,465	309,409,282	0
Average Latency	15 cycles	13 cycles	13 cycles
Average BW Utilization	0.526	3.456	0.020
CPI Rate	0.766	1.258	0.680
CPU Utilization	24.3%	95.4%	43.0%
Cache-Misses(all cache refs)	6.145%	10.857%	5.835%

TABLE VIII: Profiling for Dataset 2

Metrics	2 Threads	3 Threads	4 Threads
Memory Bound(Pipeline Slots)	75.3%	72.6%	73.7%
Cache Bound(Clockticks)	57.6%	57.1%	58.7%
DRAM Bound(Clockticks)	30.8%	28.7%	27.5%
LLC Miss Counts	172,905,187	344,410,332	309,409,282
Average Latency	15 cycles	15 cycles	13 cycles
Average BW Utilization	2.293	2.658	3.456
CPI Rate	0.980	1.087	1.258
CPU Utilization	24.3%	70.9%	95.4%
Cache-Misses(all cache refs)	10.544%	10.223%	10.857%

TABLE IX: Profiling of Parallel Implementation using different number of threads for Dataset 2

Memory bound refers to a situation in which the time to complete a given computational problem is decided primarily by the amount of memory required to hold the working data. Thus, we can improve the memory bound by parallelization as we decrease the access overhead by better restructured reads. The State of the Art Implementation has far better memory bound than our implementation since it uses map data structure for storage. It has less number of cache-misses and nearly zero last level-cache misses.

Cache Bound metric shows how often the machine was stalled on L1, L2, and L3 caches. We were able to decrease the cache bound by 12.7%, and tried to reach closer to that of the State of the Art. The cache bound is better for state of the art algorithm because it utilizes lesser memory due to application of the map data structure.

For parallel implementation we observe better CPU and Bandwidth Utilization. Also the average latency reduces in parallel implementation.

The number of cache misses in parallel implementations is higher than in serial implementations. One explanation for this is that, because multiple cores are working on various instructions, there is a very high chance that one core will discard an instruction that another core would require. As a result, for parallel implementation, the cache misses are larger.

B. Cache Miss Analysis

For cache miss analysis, we used the valgrind tool [14]. Valgrind is a cache profiler with a lot of features. For cache analysis, we used the valgrind tool cachegrind. We performed the analysis for our own parallel algorithm implementation on Dataset2.

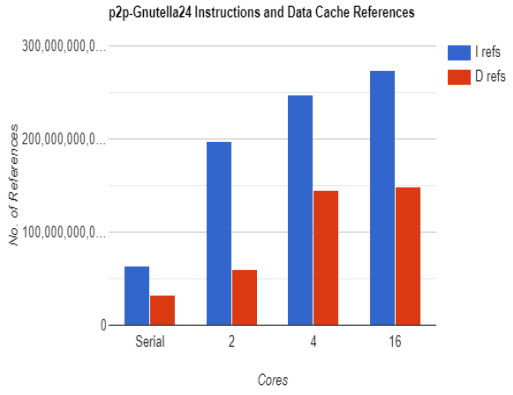

Fig. 9: Instruction and Data Cache Refs for Dataset2

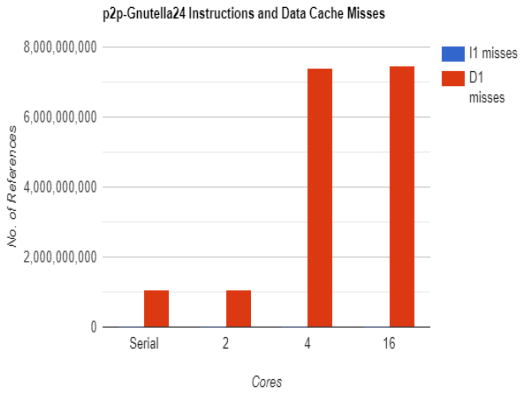

Fig. 10: Instruction and Data Cache Misses for Dataset2

Cores	I1 Miss Rate	D1 Miss Rate
Serial	$\approx 0.0\%$	3.3%
2	$\approx 0.0\%$	1.8%
4	$\approx 0.0\%$	5.1%
16	$\approx 0.0\%$	5.0%

TABLE X: I1 and D1 Miss Rates for Dataset2

REFERENCES

[1] 'PageRank'. Wikipedia, 8 Feb. 2022. Wikipedia, https://en.wikipedia.org/w/index.php?title=PageRankoldid=1070641114

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

[3] Dubey, H., Roy, B. N. (2011). An improved page rank algorithm based on optimized normalization technique.

[4] Zhu Y, Ye S, Li X. Distributed pagerank computation based on iterative aggregation-disaggregation methods. ACM, Bremen, Germany: Proceedings of the 14th ACM international conference on Information and knowledge management. 2005 Oct; 578-85.

[5] Wu T, Wang B, Shan Y, Yan F, Wang Y, Xu N. Efficient PageRank and SpMV Computation on AMD GPUs. San Diego, CA: IEEE 39th International Conference on Parallel Processing. 2010 Sep; p. 81-9.

[6] Cevahir A. Aykanat C. Turk A, Cambazoglu BB, Nukada A, Matsuoka S. Efficient PageRank on GPU clusters. IPSJ SIG Technical Report, HPC-128. 2010.

[7] Sankaralingam K, Sethummadhavan S, Browne JC. Distributed PageRank for P2P Systems. Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing. 2003 June; p. 58-68.

[8] Cevahir A, Aykanat C, Turk A, Cambazoglu BB. Site-Based Partitioning and Repartitioning Techniques for Parallel PageRank Computation. IEEE Transactions on Parallel and Distributed Systems. 2011 May; 22(5):786-802.

[9] Xing, W., Ghorbani, A. (2004, May). Weighted pagerank algorithm. In Proceedings. Second Annual Conference on Communication Networks and Services Research, 2004. (pp. 305-314). IEEE.

[10] http://snap.stanford.edu/data/index.html

[11] https://davideliu.com/2020/02/27/analysis-of-parallel-version-of-pagerank-algorithm/

[12] State of the Art Implementation, https://github.com/davide97l/PageRank-Parallel

[13] Time Complexity of Adjacency List https://stackoverflow.com/questions/44983431/time-complexity-of-adjacency-list-representation

[14] Valgrind https://valgrind.org/docs/manual/quick-start.html

[15] B. van Strien, K. Rietveld and H. Wijshoff, "Deriving Highly Efficient Implementations of Parallel PageRank," 2017 46th International Conference on Parallel Processing Workshops (ICPPW), 2017, pp. 95-102, doi: 10.1109/ICPPW.2017.26.

[16] H. Eedi, S. Peri, N. Ranabothu and R. Utkoor, "An Efficient Practical Non-Blocking PageRank Algorithm for Large Scale Graphs," 2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2021, pp. 35-43, doi: 10.1109/PDP52278.2021.00015.