EE451 Project Report

Parallel Page Rank: Project Report

Teammates: Deep Sheth (9121441038) and Nidhish Sawant (4391080611)

Introduction

The PageRank algorithm is a cornerstone in network analysis and has wide applications in web search, social network analysis, and information retrieval. It calculates the probability distribution representing the likelihood of a person randomly clicking on links arriving at a particular page. Unlike simple vote counting, PageRank accounts for the prestige of the voters, ensuring a democratic but weighted assessment of web pages.
This project explores the implementation and optimization of the PageRank algorithm in a parallel computing environment. By leveraging advanced computational techniques, we aim to improve performance, especially for large-scale datasets.

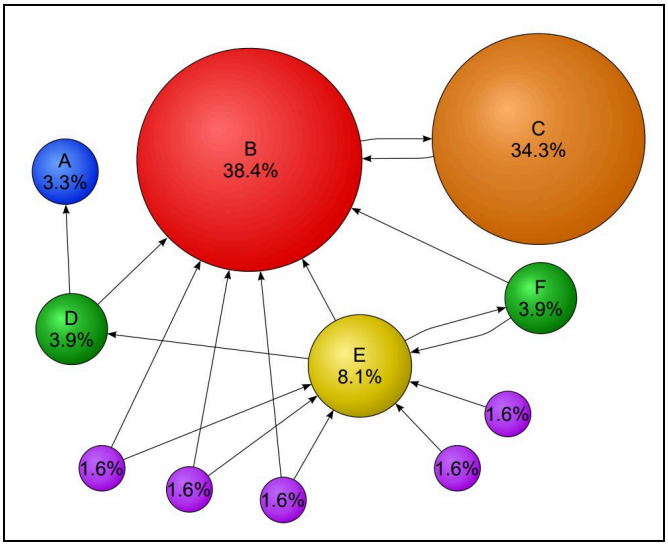

Figure 1. Example of Page Ranks

Hypothesis

1. The Message Passing Parallel Implementation for the PageRank algorithm takes more time than the CUDA programming implementation.
2. The Message Passing Algorithm requires more time than the serial implementation of the PageRank algorithm.

Algorithm

PageRank assigns a rank P(i) to a page i based on the ranks of pages linking to it:

$P(i) = \Sigma(P(j) / O\_j)$ for all j pointing to i,

where O_j is the number of outgoing links from page j. Initially, all pages are assigned an equal rank. The algorithm iteratively updates these ranks until they converge to stable values.

The serial implementation involves:
1. Initializing ranks.
2. Iterative rank updates using matrix operations until convergence.

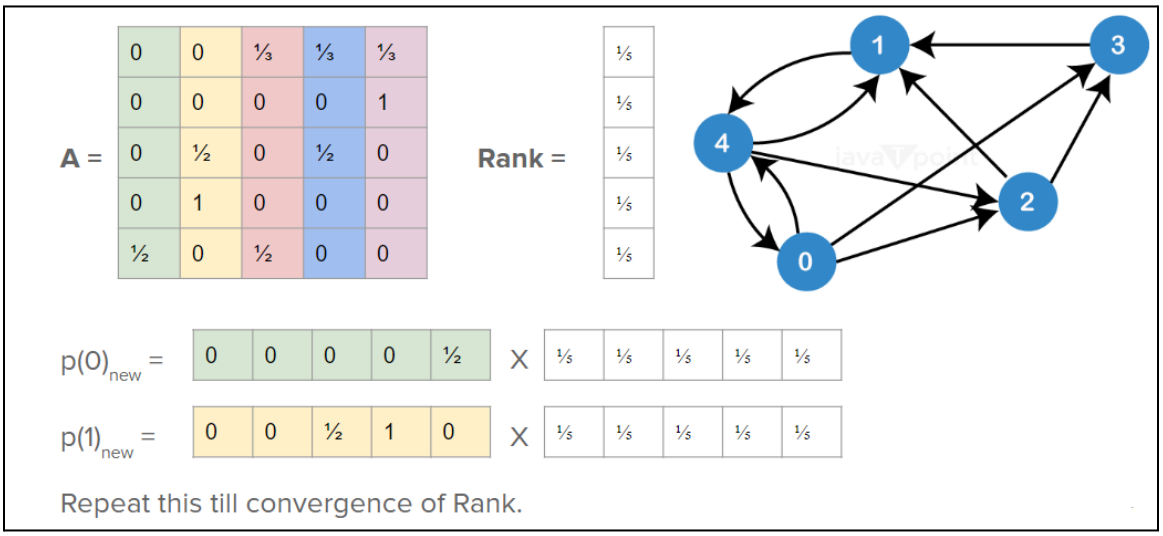

Figure 2. Visual explanation of PageRank Algorithm

Dataset

The datasets used in this study include:

- Wiki-Vote dataset: Represents Wikipedia administrators' voting history (January 2008), with 7,115 nodes and 103,689 edges.
- Datasets such as Google Graph (nodes: 875,713 edges: 5,105,039) and Wiki Talk Graph are used (nodes: 2,394,385 edges: 5,021,410) as extensions for testing scalability.

Datasets	No. of Nodes	No. of Edges
Wiki-Vote	7,115	103,689
Wiki-Talk	2,394,385	5,021,410
Web-Google	875,713	5,105,039

Context

PageRank's reliance on adjacency matrices for graph representation introduces significant memory challenges, particularly when dealing with large, sparse graphs. Traditional dense matrix formats lead to high memory overheads, while sparse representations help reduce storage and improve performance. However, even with efficient representations, the computational demands remain high.

In distributed parallel implementations using MPI, processes must frequently exchange PageRank values corresponding to edges that cross process boundaries. These exchanges can cause communication overhead and potential load imbalances across the computational nodes. Optimizing this communication pattern—either through more clever partitioning or communication overlap—can significantly improve scalability.

Furthermore, the shift towards GPU computing offers an opportunity to accelerate PageRank computations. Leveraging CUDA, developers can exploit the massive parallelism available on modern GPUs to perform vector-matrix operations—core components of the PageRank update step—much more quickly than on a CPU. However, harnessing this potential comes with its own challenges:

- **Memory Transfer Overheads:** Moving large adjacency data structures and rank vectors between host (CPU) and device (GPU) memory can be costly.
- **Efficient Parallelization:** Mapping PageRank's iterative update step to GPU kernels requires careful organization of threads and blocks, as well as attention to shared memory usage and coalesced memory accesses to maximize throughput.
- **Load Balancing and Irregularity:** Irregular graph structures, where some nodes have far more incoming or outgoing links than others, can lead to imbalanced workloads on the GPU. Implementing techniques like warp-level synchronization and intelligent partitioning can help mitigate these issues.

In summary, while MPI-based parallelization addresses scalability across multiple nodes, CUDA-based acceleration tackles single-node performance, capitalizing on GPUs' parallelism.

Together, these techniques pave the way for efficiently handling very large graphs and bringing PageRank computations closer to real-time performance requirements.

Serial Code

Adjacency Matrix

This code implements the PageRank algorithm in a serial manner. It first reads a set of edges from a file, stores them, and uses a map to identify all unique nodes. Instead of relying on the maximum node ID to determine the number of nodes, it assigns each unique node a new ID, creating a continuous ID space from 0 to n-1. Once all nodes are identified and edges are re-mapped to these new IDs, the code constructs a corresponding adjacency matrix.

After building and normalizing the adjacency matrix (and handling dangling nodes), it computes the PageRank vector iteratively until convergence or a set number of iterations. Each iteration involves multiplying the transpose of the adjacency matrix by the current rank vector, adding the damping factor, and checking the error for convergence. In the end, it prints out the number of iterations, the first 10 PageRank values, and the computation time**.**

Limitations in Adjacency Matrix form

The current implementation faces challenges such as:
- Memory inefficiencies with dense adjacency matrices.

Adjacency List

This program implements a serial version of the PageRank algorithm. It begins by reading an edge list from a specified input file. Each line in the file represents a directed edge in the form "from_idx to_idx". As the program reads these edges, it dynamically assigns a unique integer ID to each newly encountered node, building a mapping between the node's original identifier and its new zero-based index.

Once all edges are read, the code constructs a list of Page structures, where each Page holds information such as incoming links, the count of incoming and outgoing edges, and a placeholder for its eventual PageRank score. Using this information, the code sets up arrays to track out-degree counts and their reciprocals, identifies dangling pages (nodes with no outlinks), and initializes the PageRank vector with equal probabilities.

The PageRank computation proceeds iteratively. In each iteration:

1. AddPagesPr: Incorporates contributions from each page's incoming neighbors, weighted by their outgoing link counts.
2. AddDanglingPagesPr: Accounts for pages with no outlinks, distributing their PageRank evenly among all nodes.

3. AddRandomJumpsPr: Implements the random surfing model, where a fraction of the rank is redistributed equally among all pages to ensure convergence and handle disconnected components.

After a fixed number of iterations (default 80), the code prints the final PageRank values for the first 10 pages and reports the total computation time. This approach is straightforward and illustrative, but it relies on a dense representation of the graph and runs in a single process/thread, which can limit scalability for very large graphs.

Parallel Implementation

MPI

This code implements a parallel PageRank computation using MPI. It reads a graph from an edge list file, assigning unique IDs to nodes as they are encountered. Once the nodes and their incoming links are established, the code sets up data structures to store the graph, including a mapping from original to internal node IDs and a record of each node's incoming edges and out-degree.

The PageRank calculation proceeds in a distributed manner across multiple MPI processes. Each process handles a segment (chunk) of the node range. The algorithm iterates multiple times, and in each iteration:

1. **Broadcast Current Ranks:** The root process broadcasts the current PageRank vector to all other processes so that every process can have the up-to-date ranks for all nodes.
2. **Partial Computation (AddPagesPr):** Each process computes partial contributions to PageRank for the subset of nodes it is responsible for, using only the portion of the graph assigned to it.
3. **Allgather Results:** After local computations, all processes gather their updated partial PageRank values into a global PageRank vector.
4. **Final Adjustments at Root:** The root process handles any remainder nodes that don't fit evenly into partitions, applies dangling page adjustments, and adds random jumps to complete the iteration's PageRank update.

This MPI-based approach allows the PageRank computation to scale to larger graphs by distributing both data and computation among multiple processes, potentially improving performance and enabling the handling of more extensive datasets than would be feasible with a single-process implementation.

CUDA

This code is a CUDA-accelerated implementation of the PageRank algorithm. It begins by reading a graph from an input file, where each line specifies a directed edge between two nodes. As it reads the edges, it assigns each newly encountered node a unique ID and stores incoming link information. After determining the total number of nodes and constructing data structures

that represent the graph (including a flattened list of incoming edges and their start positions), the code initializes a PageRank vector with equal probability for each node.

The PageRank calculation is performed on the GPU. Each iteration involves three main steps implemented as CUDA kernels:

1. **AddPagesPrKernel:** Computes the new PageRank values based on incoming links and their corresponding old PageRank scores.
2. **SumDanglingKernel + AddDanglingPagesPrKernel:** Handles pages with no outlinks (dangling pages) by distributing their rank evenly among all nodes.
3. **AddRandomJumpsPrKernel:** Adds a "random jump" factor, ensuring that the rank distribution doesn't get stuck and simulates the behavior of a random surfer.

By offloading the core computations (like matrix-vector multiplications) to the GPU, this approach leverages CUDA's parallelism to potentially achieve significant speedups over a pure CPU implementation.

Results

DataSets	Serial	Serial- Edge List	MPI - 2 threads	MPI - 4 threads	MPI - 8 threads	Cuda
Wiki-Vote	9351.24	72.0758	121.329	95.1724	74.4033	13.4464
Wiki-Talk	NA	11416.3	12363.7	11681.76	10728.8	483.4
Web-Google	NA	9879.99	10492	8295.5	6838.47	241.359

Table 1. Runtime of algorithms in milliseconds.

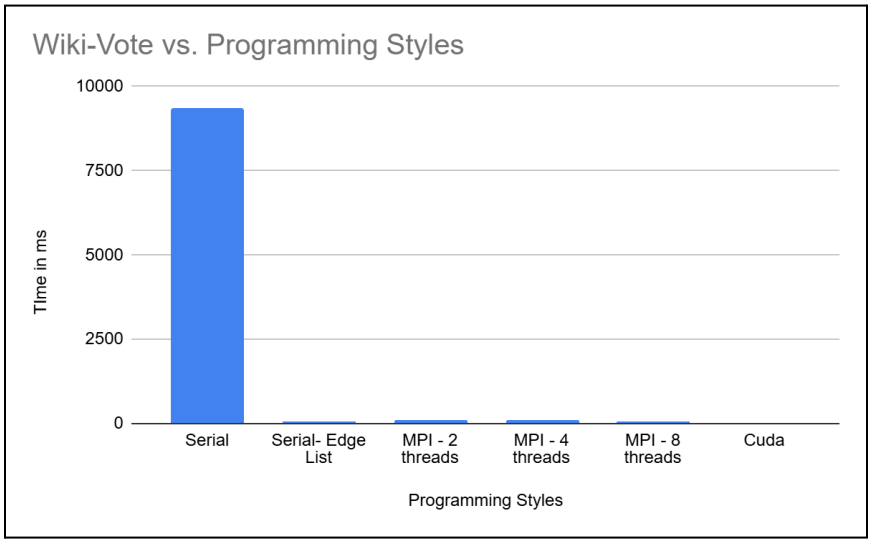

Figure 3. Runtime for Wiki-Vote dataset in milliseconds.

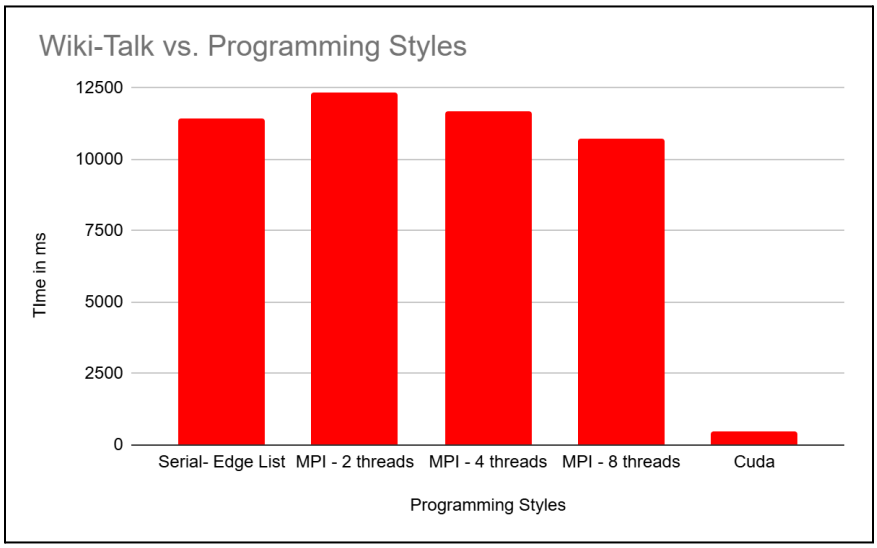

Figure 4. Runtime for Wiki-Talk dataset in milliseconds.

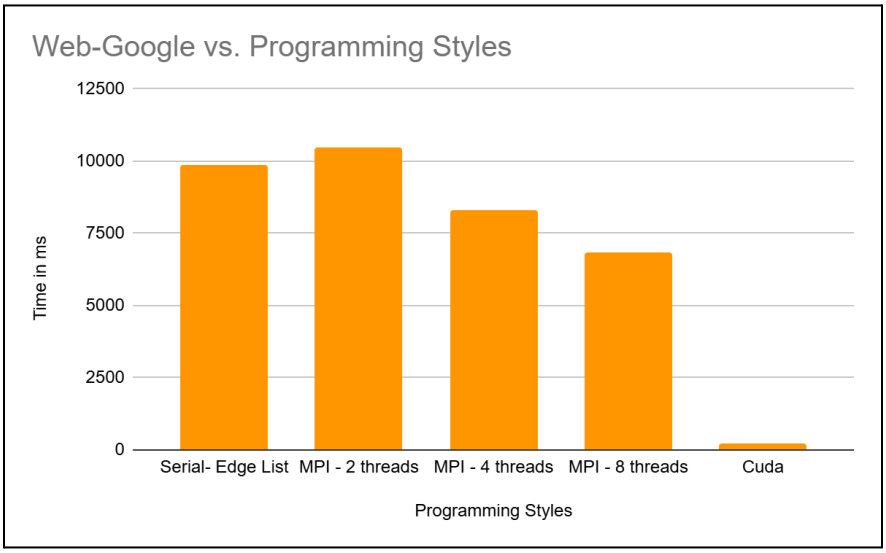

Figure 5. Runtime for Wiki-Talk dataset in milliseconds.

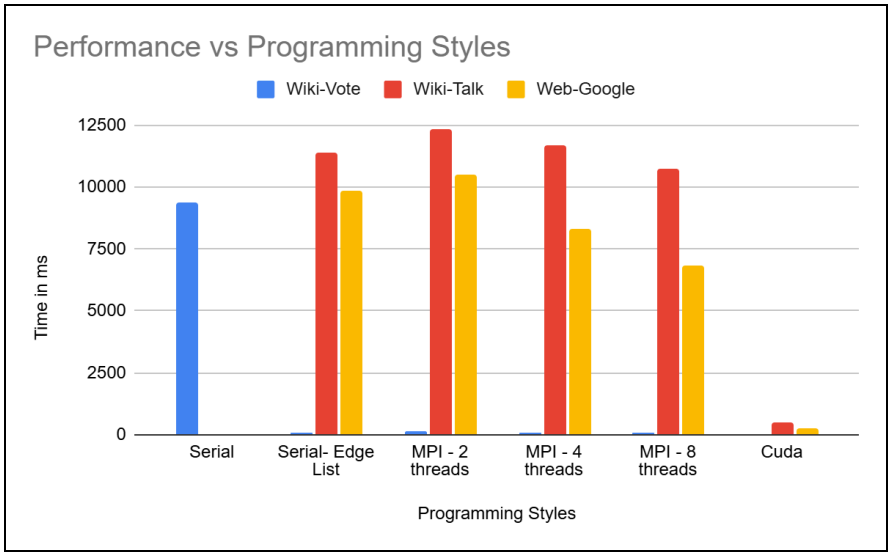

Figure 3. Runtime comparison for all datasets in milliseconds.

Conclusion

The project results strongly validate the stated hypothesis:

1. **CUDA Programming Implementation is Faster than MPI**
   The CUDA implementation consistently outperformed MPI in terms of execution time across all datasets. For example, in the Wiki-Talk dataset, the CUDA implementation completed the PageRank calculations in just 10728.8 milliseconds compared to 483.4 milliseconds for MPI with 8 threads. A speedup by a factor of 22.19. This demonstrates the superior computational throughput of GPUs. The reduced overhead in memory access and efficient parallelism provided by CUDA kernels make it a more efficient option for PageRank computations.
2. **Message Passing Parallel Implementation Takes More Time than the Serial Version**
   For smaller datasets, the overhead of message passing in the MPI implementation outweighed the benefits of parallelism, resulting in longer execution times compared to the serial implementation. For instance, the MPI implementation with 2 threads took 12363.7 milliseconds for the Wiki-Talk dataset, while the serial edge list implementation completed in 11416.3 milliseconds. This behavior aligns with the hypothesis, as the communication delays and load-balancing issues in MPI become significant bottlenecks, especially for datasets that are not large enough to fully utilize distributed computing resources. But for larger datasets increasing the number of threads give better performance than the serial implementation (#threads=8).

These observations validate that while parallel implementations improve scalability and enable handling larger datasets, the choice of parallelization technique (MPI or CUDA) significantly impacts performance. CUDA emerges as the superior option for large-scale graph computations, proving the first part of the hypothesis, while the second part is confirmed by the MPI implementation's slower performance compared to serial for smaller datasets.