Page Rank

Luka Svensek
89231433@student.upr.si

ABSTRACT

This project explores the implementation of the PageRank algorithm, a foundational method for ranking the importance of nodes within a linked database, such as the World Wide Web. We compare three distinct execution models: sequential, parallel, and distributed. The objective is to analyze the advantages and disadvantages of each approach concerning performance, scalability, and implementation complexity. The sequential version provides a baseline for comparison, the parallel version leverages a multi-core processor for enhanced speed, and the distributed version partitions the task across multiple independent processes using MPI.

ACM Reference Format:
Luka Svensek . 2025. Page Rank. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Keywords: PageRank, Parallel Computing, Distributed Systems, MPI, Java, Graph Algorithms

1 INTRODUCTION

The PageRank algorithm, developed by Google's co-founders, offers a solution to determine a webpage's importance based on its link structure. It interprets a hyperlink from page A to page B as a vote of confidence, where votes from more important pages carry more weight. The algorithm also incorporates a damping factor, which simulates a user's tendency to either follow links or jump to a random page. This project focuses on implementing this algorithm to understand the practical trade-offs between different computational paradigms.

2 THE PAGERANK ALGORITHM

The core of the PageRank algorithm is an iterative process that updates the rank of each page based on the ranks of the pages that link to it. The rank of a page is calculated using the following formula:

$$PR(i) = \frac{1-d}{N} + d \sum_{j \in M(i)} \frac{PR(j)}{L(j)} \quad (1)$$

Where:

- $PR(i)$ is the PageRank of page i.
- N is the total number of pages (vertices) in the graph.
- d is the damping factor, a value typically set around 0.85.
- $M(i)$ is the set of pages that link to page i.
- $L(j)$ is the number of outbound links from page j.

The process is repeated until the ranks converge, meaning the changes between iterations fall below a small threshold.

3 SYSTEM ARCHITECTURE AND DATA REPRESENTATION

The project is built in Java and structured into three main classes: `Main.java`, `PageRank.java`, and `Distributed.java`. While the project guidelines recommended using an adjacency list, this implementation utilizes a **boolean adjacency matrix** (`boolean[][] adj`). In this matrix, `adj[i][j]` is true if a directed edge exists from node i to node j. This choice offers $O(1)$ lookup time for edge existence at the cost of $O(V^2)$ memory usage.

4 SEQUENTIAL IMPLEMENTATION

The sequential implementation serves as the foundational approach. All computations are executed in a single thread. In each iteration, the program calculates a new rank for every vertex by iterating through the entire graph. While this method is the most straightforward to implement and debug, its performance is limited by the processing power of a single CPU core, making it unsuitable for large-scale simulations.

5 PARALLEL IMPLEMENTATION

To improve performance on modern hardware, a parallel version was developed. This implementation leverages Java 8's Parallel Streams API (`java.util.stream.IntStream.range(0, n).parallel()`) to distribute the workload of the main calculation loop across multiple CPU cores. This allows the ranks of different vertices to be computed concurrently. The implementation automatically adapts to the available hardware, as the underlying framework manages the thread pool. This approach significantly enhances performance for large graphs on a single machine.

6 DISTRIBUTED IMPLEMENTATION

The distributed version is designed for maximum scalability, allowing the computation to be spread across multiple machines or processes. It uses the MPJ Express library, a Java implementation of the Message Passing Interface (MPI) standard. The root process (rank 0) generates the graph and broadcasts it to all worker processes. The workload is then partitioned, with each process calculating a subset of the new ranks. An `Allreduce` collective operation is used to sum the partial results and distribute the final, updated rank vector back to all processes for the next iteration. While this offers the greatest potential for handling massive datasets, it introduces the overhead of network communication.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

7 PERFORMANCE ANALYSIS AND GUIDELINE ADHERENCE

A key guideline was to measure the total runtime and the time per iteration. This feature was not implemented in the provided code, so a direct quantitative comparison is not possible. However, we can analyze the expected performance:

- **Sequential:** Slowest, serving as the performance baseline.
- **Parallel:** Significant speedup over sequential, limited by the number of cores on a single machine.
- **Distributed:** Theoretically the most scalable, but performance is sensitive to network latency and the overhead of data serialization and communication.

The implementation adheres to most guidelines, including mode selection, parameter input, random graph generation with a seed, and CSV output. The primary deviations are the use of an adjacency matrix instead of an adjacency list and the lack of runtime measurement.

8 FUTURE WORK AND OPTIMIZATIONS

Several improvements could be made in future iterations:

(1) **Implement Runtime Measurement:** Add precise timing for total execution and per-iteration performance to allow for quantitative analysis.
(2) **Refactor to Adjacency List:** Change the graph representation to an adjacency list to reduce memory consumption, which would be more efficient for large, sparse graphs.
(3) **GPU Acceleration:** For the parallel version, explore using a library like Aparapi to offload the computation to a GPU for a massive performance boost.

9 CONCLUSION

This project successfully demonstrates that the PageRank algorithm can be implemented using sequential, parallel, and distributed approaches. The sequential version is a valuable educational baseline, the parallel implementation offers a practical performance boost on common hardware, and the distributed solution provides a path toward massive scalability. The project highlights the fundamental trade-offs between algorithm complexity, hardware utilization, and performance in the field of high-performance computing.