Parallelizing PageRank with OpenMP, CUDA, and MPI-CUDA

Ananya Mantravadi
amantra@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Abstract

Page Rank is a graph algorithm developed by Google to rank web pages based on their importance. It is used in web search engines, social network analysis, and recommendation systems. The algorithm models the web as a directed graph, where each node represents a webpage, and edges represent hyperlinks between them. PageRank assigns a ranking score to each node based on the probability of a random web surfer visiting it. We evaluate variants using dense adjacency matrices to focus on the impact of different parallelization strategies: a sequential baseline, a shared-memory OpenMP version, a CUDA GPU-accelerated version, and a hybrid MPI + CUDA version for distributed multi-GPU environments.

Keywords

PageRank, Profiling, Hybrid Parallelism, OpenMP, MPI, CUDA

ACM Reference Format:
Ananya Mantravadi. 2025. Parallelizing PageRank with OpenMP, CUDA, and MPI-CUDA. In *Proceedings of Parallel Systems (CSC 548).* ACM, New York, NY, USA, 5 pages.

1 Introduction

The explosive growth of the Internet has led to the need for intelligent ranking algorithms to efficiently retrieve relevant information. One of the most influential such algorithms is **PageRank**, originally developed by Google founders Larry Page and Sergey Brin [2]. As discussed in [1], PageRank models the web as a directed graph where each node represents a webpage and edges represent hyperlinks. The main idea here is that a page is important if it is linked to by other important pages. This recursive notion leads naturally to a linear algebraic formulation involving eigenvectors.

In the PageRank model, a *random surfer* navigates the web by clicking on links at random, with some probability of teleporting to a random page. Mathematically, this behavior is captured by the power iteration method applied to a stochastic transition matrix derived from the web graph. Each iteration updates the rank vector based on the ranks of the linked pages until convergence. The PageRank value of a page reflects the steady-state probability that the random surfer is at that page.

Let r be the rank vector and P the transition probability matrix of the web graph. The PageRank computation solves the eigenvector

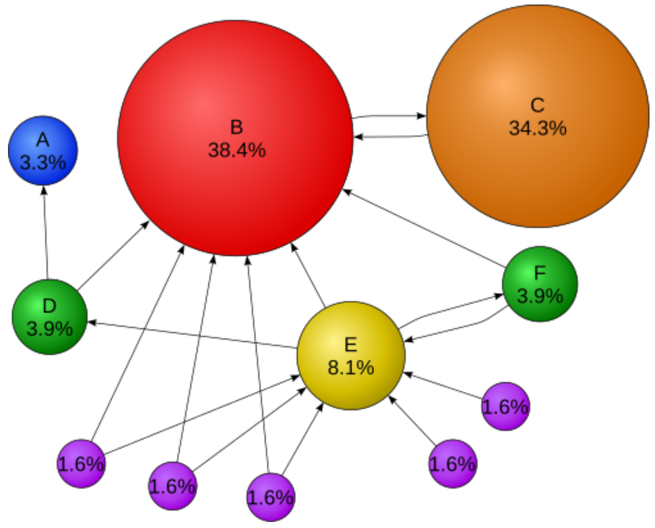

Figure 1: Pagerank algorithm illustration. The percentage shows the perceived importance, and the arrows represent hyperlinks [8])

equation:

$$r = \alpha P r + \frac{(1-\alpha)}{n} e$$

where e is the vector of all ones, n is the number of pages, and α (typically 0.85) is the damping factor representing the probability of following a hyperlink versus teleporting. This modification ensures that the matrix is stochastic and irreducible, guaranteeing the existence and uniqueness of a solution.

1.1 Illustrative Example

Consider a small web with three pages A, B, and C with this link structure:

- Page A links to Page B.
- Page B links to Pages A and C.
- Page C links to Page A.

This can be represented by the following **link matrix** L, where $L_{ij} = 1$ if page j links to page i, and 0 otherwise:

$$L = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

We then construct the **transition probability matrix** P, where each column sums to 1 by normalizing the outlinks:

$$P = \begin{bmatrix} 0 & \frac{1}{2} & 1 \\ 1 & 0 & 0 \\ 0 & \frac{1}{2} & 0 \end{bmatrix}$$

Table 1: Experimental Settings

Environment	Processor / GPU	Memory	Setup
Local Machine	Intel i5-1335U (12 threads)	16 GB RAM	Sequential, OpenMP
ARC Skylake Cluster	Intel Skylake Silver CPU	96 GB RAM	MPI (np=4, 2 nodes)
ARC A6000 Cluster	NVIDIA RTX A6000 GPU	96 GB RAM	MPI+CUDA (np=2, 2 GPUs)

To incorporate the random teleportation with damping factor $\alpha = 0.85$, the Google matrix G is defined as follows where E is a 3×3 matrix of all ones:

$$G = \alpha P + \frac{(1-\alpha)}{3} E$$

$$G = 0.85 \times \begin{bmatrix} 0 & \frac{1}{2} & 1 \\ 1 & 0 & 0 \\ 0 & \frac{1}{2} & 0 \end{bmatrix} + 0.15 \times \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix}$$

$$G = \begin{bmatrix} 0.05 & 0.475 & 0.90 \\ 0.90 & 0.05 & 0.05 \\ 0.05 & 0.475 & 0.05 \end{bmatrix}$$

Now, starting from an initial rank vector $r^{(0)} = \left[\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right]^T$, we apply the power method $r^{(1)} = Gr^{(0)}$ and repeat until convergence (difference between previous and current rank vector values is less than ϵ). The final steady-state rank vector r gives the relative importance scores of pages A, B, and C.

The computation of PageRank is challenging because of the following reasons. The web graph is extremely large, often involving billions of nodes and hyperlinks. The iterative nature of the power method requires many matrix-vector multiplications. The complexity of this algorithm is $O(n^3)$, which will be very heavy to handle to scale. Sparse matrix storage and efficient memory access are critical to performance. It is highly parallelizable due to its data-parallel structure [4]. A known bottleneck is the communication and synchronization required each iteration. Even with significant speedups, inter-node synchronization overhead can limit efficiency in MPI or distributed setups. Therefore we need to ensure load balance and efficient memory access. Each page's rank update depends only on its incoming links and can be computed independently. Real-world web graphs are sparse, but in this project, we use dense matrices to simplify the implementation and focus on comparing the effectiveness of different parallelization strategies. All experiments are carried out on the network repository Web Graphs datasets [6].

2 Algorithms of Variants

We implement and evaluate the PageRank algorithm using various parallel computing models:

- A sequential CPU baseline to establish a reference point.
- An OpenMP version to exploit shared-memory multicore parallelism.
- A CUDA version that uses GPU acceleration for matrix operations.
- A hybrid MPI+CUDA version distributing computation across multiple GPUs and nodes.

Each implementation shares a common computational structure - initialization, normalization of the transition matrix, iterative power method updates, and convergence checking. The experimental settings are described in Table 1. Correctness of all implementations was validated by verifying convergence thresholds and comparing top-ranked nodes across methods for smaller input graphs.

2.1 Sequential

We use a dense adjacency matrix to represent the graph. It initializes the rank vector uniformly, normalizes the matrix columns to handle dangling nodes, and then iteratively applies the power method until convergence. In each iteration a new rank vector is computed by multiplying the transition matrix with the previous rank vector, and convergence is checked based on the difference norm.

Algorithm 1 Sequential PageRank

```
1: **Input:** Adjacency matrix $G$, damping factor $\alpha$, tolerance $\epsilon$
2: Initialize $r \leftarrow \left[\frac{1}{n}, \frac{1}{n}, \dots, \frac{1}{n}\right]$
3: Normalize columns of $G$
4: **repeat**
5:     $r_{\text{last}} \leftarrow r$
6:     **for** each node $i$ **do**
7:         $r[i] \leftarrow \sum_{j=1}^{n} G[i][j] \times r_{\text{last}}[j]$
8:     **end for**
9:     Compute difference $\Delta \leftarrow \sum_{i=1}^{n} |r[i] - r_{\text{last}}[i]|$
10: **until** $\Delta < \epsilon$
11: **Output:** Rank vector $r$
```

2.2 OpenMP

The OpenMP version parallelizes the loops in the sequential code using multithreading (12 threads). The initialization, normalization, matrix-vector multiplication, and convergence check are all performed in parallel across available CPU cores, taking advantage of shared-memory parallelism by using `#pragma omp` directives.

Algorithm 2 OpenMP Parallel PageRank

```
1: **Input:** Adjacency matrix $G$, damping factor $\alpha$, tolerance $\epsilon$
2: Initialize $r$ uniformly using parallel for loop
3: Normalize columns of $G$ using parallel for loop
4: **repeat**
5:     $r_{\text{last}} \leftarrow r$ (parallel copy)
6:     **parallel for** each node $i$
7:     $r[i] \leftarrow \sum_{j=1}^{n} G[i][j] \times r_{\text{last}}[j]$
8:     Compute $\Delta$ using parallel reduction on $|r[i] - r_{\text{last}}[i]|$
9: **until** $\Delta < \epsilon$
10: **Output:** Rank vector $r$
```

2.3 CUDA

Each node's rank update is mapped to a GPU thread. The algorithm consists of several kernels: one for initializing the rank vector, one for storing the last iteration's rank, one for matrix-vector multiplication, and one for computing the difference between iterations. Thrust library functions are used for reduction and sorting. After experimenting with different block sizes, I chose 128, which provided the least runtime. As number of threads in a block increases, performance decreases. Each thread processes one row, but row sizes are uniform, so warp divergence is minimal.

Algorithm 3 CUDA Parallel PageRank

```
1: Input: Adjacency matrix $G$ (flattened), damping factor $\alpha$, tolerance $\epsilon$
2: Launch initialize_rank kernel to set $r$ uniformly
3: Launch manage_adj_matrix kernel to normalize columns
4: repeat
5:     Launch store_rank kernel to copy $r$ to $r_{\text{last}}$
6:     Launch matmul kernel to compute new $r$
7:     Launch rank_diff kernel to compute element-wise differences
8:     Use thrust::reduce to compute $\Delta$
9: until $\Delta < \epsilon$
10: Output: Rank vector $r$
```

2.4 MPI + CUDA Hybrid

Here, multiple GPUs distributed across nodes cooperatively compute PageRank. Each MPI process uses its local GPU (2 in this project). After each iteration, a global reduction is performed using MPI_Allreduce to synchronize the convergence status across processes. This approach combines intra-node GPU parallelism with inter-node distributed memory communication.

Algorithm 4 MPI + CUDA Hybrid PageRank

```
1: Input: Adjacency matrix $G$, damping factor $\alpha$, tolerance $\epsilon$
2: Each MPI rank initializes its GPU and loads $G$
3: Launch initialize_rank and manage_adj_matrix kernels
4: repeat
5:     Launch store_rank kernel
6:     Launch matmul kernel
7:     Launch rank_diff kernel
8:     Use thrust::reduce for local difference
9:     Use MPI_Allreduce to compute global $\Delta$
10: until global $\Delta < \epsilon$
11: MPI rank 0 collects and outputs final rank vector
```

3 Results and Analysis

3.1 GPU profiling and Bottleneck analysis

We perform profiling on the CUDA implementation using NVIDIA Nsight Systems (nsys) [5], to gain more insights into kernel execution time, CUDA API overheads, and memory transfer bottlenecks. The major findings are:

- **Kernel Execution:** The `matmul` kernel (dense matrix-vector multiplication) dominated the GPU execution time, consuming approximately 92.9% of total GPU time. This matches expectations, as the power method is computationally heavy on matrix operations.
- **Memory Operations:** Significant time (around 25.8%) was spent in Host-to-Device memory transfers, indicating a notable overhead during graph initialization and copying input data. Device-to-Host transfers were minimal.
- **CUDA API Overheads:** `cudaDeviceSynchronize`, `cudaMalloc`, and `cudaMemcpy` API calls accounted for most of the API runtime. Memory allocations (`cudaMalloc`) were relatively expensive and could be optimized by pooling memory where possible.
- **Other Kernels:** Kernels such as `manage_adj_matrix`, `rank_diff`, and `store_rank` consumed very little time individually compared to `matmul`.

The profiling results confirm that PageRank's performance on GPUs is compute-bound, dominated by matrix-vector multiplications. Efficient kernel optimization and memory transfer reduction are key opportunities for further acceleration.

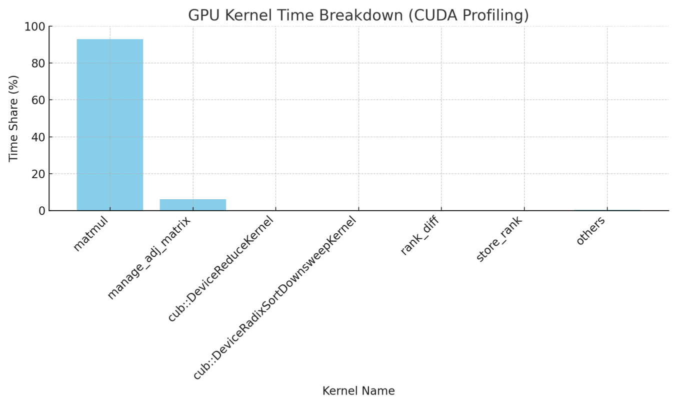

Figure 3: GPU Kernel Time Breakdown

3.2 Performance comparison

One thing to note is that there is no trend with respect to number of nodes. Performance of PageRank algorithm also depends on how nodes are connected too. If it's a perfectly reducible matrix, algorithm converges quickly, but if that's not the case, algorithm doesn't converge even after 1000 iterations. So we cannot predict the outcome strictly by the number of nodes, but rather, we have to base it on the underlying connections/links of nodes within the graph.

Figure 2 shows the runtime comparison in milliseconds across Sequential, OpenMP, CUDA, and MPI+CUDA implementations for different graph sizes. A logarithmic scale is used for the time axis to clearly distinguish performance differences. Table 2 shows the overall speedup observed over the sequential baseline.

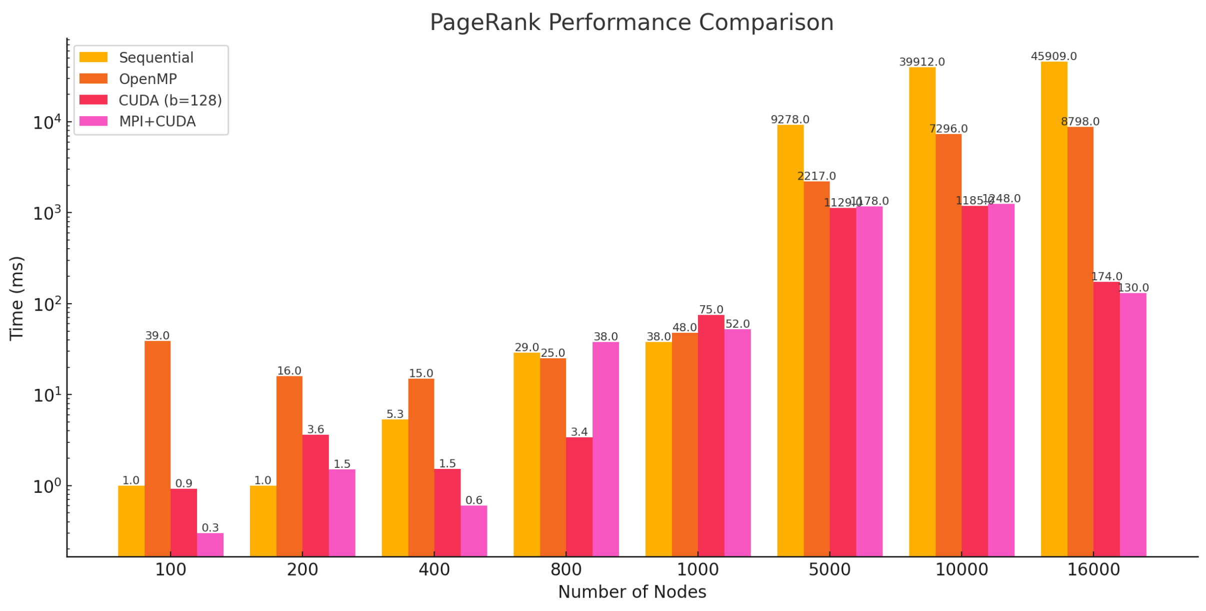

Figure 2: Performance Comparison

Table 2: Speedup over Sequential Implementation

Nodes	OpenMP	CUDA	MPI+CUDA
100	0.03×	1.09×	3.33×
200	0.06×	0.28×	0.67×
400	0.35×	3.49×	8.83×
800	1.16×	8.60×	0.76×
1000	0.79×	0.51×	0.73×
5000	4.18×	8.22×	7.88×
10000	5.47×	33.68×	31.98×
16000	5.22×	263.84×	353.15×

3.3 Observations

If the graph is highly connected or reducible, the power method converges faster, resulting in shorter runtimes even for larger graphs. Otherwise if the graph is poorly connected or sparse, convergence can be slow, leading to higher runtimes even for smaller node counts. The sequential implementation becomes impractical as graph size and complexity grow, with runtimes reaching over 45 seconds for 16,000 nodes graph.

It is important to note that the different implementations were executed on different hardware platforms. The sequential and OpenMP versions ran on a laptop-grade Intel Core i5 CPU, whereas the CUDA and MPI+CUDA implementations utilized powerful server-grade NVIDIA A6000 GPUs on a high-performance computing cluster. Although the trends in acceleration are clear, direct runtime comparisons must be interpreted cautiously. The results primarily highlight how well each parallelization strategy scales relative to its respective platform, rather than providing an absolute cross-platform comparison. Nevertheless, the experiments demonstrate how GPU-accelerated and distributed approaches offer substantial benefits for large-scale PageRank computations.

- **OpenMP Acceleration:** OpenMP offers noticeable speedup over the sequential version for smaller graphs, but its advantage diminishes for larger graphs due to limited memory bandwidth and synchronization overhead across CPU threads.
- **CUDA Acceleration:** CUDA dramatically reduces runtime, leveraging the massive parallelism of GPUs. For larger graphs (e.g., 16,000 nodes), CUDA runtime is two orders of magnitude lower than the sequential version. The excellent performance of CUDA in our study matches the findings of Visali et al. [7], Duong et al. [3], among others.
- **MPI + CUDA Scalability:** The hybrid MPI+CUDA implementation achieves the fastest results for large graphs with 350 times speedup for 16000 nodes graph by combining distributed GPU computation and global synchronization through MPI. Although this implementation offers distributed GPU parallelism, it did not consistently outperform the standalone CUDA version in practice. We can observe that in smaller graphs, the communication synchronization overhead introduced by MPI across nodes outweighs the benefits. Each MPI rank maintains a copy of the PageRank vector and performs local GPU computations. After each iteration, global convergence is checked using an `MPI_Allreduce` operation. Although the per-iteration GPU computation is fast, the overhead of repeatedly broadcasting and reducing across processes can dominate the runtime when the number of nodes is small (underutilizing each GPU), or when the number of MPI ranks is large (increasing communication frequency). As more MPI ranks are used, memory access patterns and kernel launches become fragmented across GPUs, leading to potential device underutilization. The expected scalability in distributed systems follows strong scaling: ideally, doubling

the number of GPUs halves the runtime. However, in reality, communication overhead, load imbalance, and Amdahl's Law limit this benefit. Increasing the number of ranks beyond that point led to marginal or even negative returns due to increased synchronization cost and idle GPU time. Future work could address these limitations using overlapping communication with computation (e.g., `MPI_Isend` / `MPI_Irecv`) and better partitioning strategies for load balancing across GPUs.

4 Limitations, Conclusion, and Future Work

The use of dense adjacency matrices across all implementations allowed for uniform and straightforward comparisons between sequential, OpenMP, CUDA, and MPI+CUDA versions, but it does not reflect the sparsity typical of real-world graphs such as web crawls or social networks. Additionally, the parallel CUDA implementation assumes uniform workloads per node, which may not hold for graphs with skewed degree distributions. Communication overhead in MPI also limited scalability, particularly on small graphs or when increasing the number of ranks without proportionally larger workloads.

This project demonstrated the design, implementation, and performance evaluation of PageRank using multiple parallel paradigms. GPU acceleration (CUDA and MPI+CUDA) showed significant speedups compared to CPU-based methods, particularly for large graphs. Profiling confirmed that the algorithm is compute-bound, dominated by matrix-vector operations. We also noted that performance is influenced not just by graph size but also by the structure and convergence behavior of the graph.

Future work will focus on sparse matrix representations such as CSR or COO to improve memory efficiency and better reflect real-world scenarios. Optimizing SpMV for sparse data on GPU and distributed memory systems could yield substantial performance improvements. Incorporating CUDA streams for asynchronous memory transfer and kernel execution may further reduce GPU idle time. On the distributed side, improvements such as non-blocking MPI communication and dynamic load balancing would improve scalability across nodes. Integrating convergence-tuning heuristics would allow more realistic evaluation of PageRank at web scale.

References

[1] David Austin. 2006. How Google Finds Your Needle in the Web's Haystack. https://www.ams.org/publicoutreach/feature-column/fcarc-pagerank. Accessed: 2025-04-30.

[2] Neelam Duhan, AK Sharma, and Komal Kumar Bhatia. 2009. Page ranking algorithms: a survey. In *2009 IEEE International Advance Computing Conference*. IEEE, 1530–1537.

[3] Nhat Tan Duong, Quang Anh Pham Nguyen, Anh Tu Nguyen, and Huu-Duc Nguyen. 2012. Parallel pagerank computation using gpus. In *Proceedings of the 3rd Symposium on Information and Communication Technology*. 223–230.

[4] Christian Kohlschütter, Paul Alexandru Chirita, and Wolfgang Nejdl. 2006. Efficient parallel computation of pageRank. In *Advances in Information Retrieval (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics))*. Springer Verlag, Germany, 241–252. doi:10.1007/11735106_22 28th European Conference on Information Retrieval Research, ECIR 2006 ; Conference date: 10-04-2006 Through 12-04-2006.

[5] NVIDIA Corporation. 2024. Nsight Systems. https://developer.nvidia.com/nsight-systems. Accessed: 2025-04-30.

[6] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com

[7] Visali V S, Manimegalai R, Noor Mahammad S K, and Sunitha Nandhini A. 2024. Performance Analysis of Parallelized PageRank Algorithm using OpenMP, MPI and CUDA. In *2024 International Conference on Smart Systems for Electrical, Electronics, Communication and Computer Engineering (ICSSEECC)*. 44–49. doi:10.1109/ICSSEECC61126.2024.10649542

[8] Wikipedia contributors. 2025. PageRank — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=PageRank&oldid=1285731222 [Online; accessed 28-April-2025].