Parallel PageRank Analysis

Using OpenMP on the SNAP web-Stanford Dataset

Course: CS-478 — Design and Analysis of Algorithms

Submitted By:
Abdullah Noor - 2022029
Ali Vijdaan - 2022560
Hamza Amin - 2022378

December 24, 2025

Abstract

This report documents the design, implementation, and evaluation of a parallel PageRank algorithm using OpenMP on a Shared Memory architecture. We utilized the `web-Stanford` dataset from the SNAP repository to evaluate performance on a real-world, scale-free graph. The solution implements a "Pull-based" synchronous iterative method with dynamic scheduling to mitigate load imbalance caused by power-law degree distributions. Experimental results on a multi-core system demonstrate a peak speedup of **2.31x** at 4 threads. However, performance degrades at higher thread counts due to the graph's high sparsity and low arithmetic intensity, which exposes the "Memory Wall" bottleneck. This report analyzes these findings using the Work-Span complexity model.

Contents

1 Introduction

1.1 Background

Graph processing is critical for applications ranging from social network analysis to web search. Sequential algorithms for problems like PageRank operate in $O(k \cdot (|V| + |E|))$ time. For massive web graphs, this latency is unacceptable.

PageRank is an iterative algorithm that measures node importance based on link structure. The core operation is a Sparse Matrix-Vector Multiplication (SpMV), which is notoriously difficult to parallelize efficiently due to irregular memory access patterns (pointer chasing) and poor cache locality.

1.2 Objectives

1. **Algorithmic Design:** Implement a lock-free parallel PageRank using OpenMP.
2. **Complexity Analysis:** Analyze the Work and Span of the parallel approach.
3. **Evaluation:** Benchmark Speedup (S_p) on the SNAP web-Stanford dataset.

2 Literature Review

The transition from sequential to parallel graph processing has been extensively studied in high-performance computing. The PageRank algorithm, originally proposed by Page et al. [1] for the Google search engine, relies on the principal eigenvector of the stochastic transition matrix of the web graph. While mathematically elegant, its computational cost on massive, sparse datasets is prohibitive.

2.1 Challenges in Parallelism

Lumsdaine et al. [3] argue that graph algorithms exhibit distinct characteristics compared to traditional dense matrix operations (e.g., LINPACK). They possess a low ratio of computation to memory access, known as low *Arithmetic Intensity*. This leads to the "Memory Wall" problem, where performance is bounded by memory bandwidth rather than CPU clock speed. This fundamental limitation explains why graph algorithms rarely achieve peak FLOPs on modern hardware.

2.2 Structural Implications and Optimization

Furthermore, the structural properties of real-world graphs complicate parallelization. Leskovec et al. [2] demonstrated that web graphs (like the `web-Stanford` dataset) follow a Power-Law (Zipfian) degree distribution. This irregularity causes severe load imbalance if static partitioning is used, as a few "hub" nodes require significantly more processing time than others.

To address synchronization overhead, Beamer et al. [4] analyzed "Push" versus "Pull" traversal strategies. They concluded that for Shared Memory architectures, a **Pull-based approach** (where nodes read from incoming neighbors) eliminates the need for atomic locks or critical sections during updates. This insight forms the basis of our lock-free OpenMP implementation.

3 Dataset Analysis: web-Stanford

For this project, we utilized the **web-Stanford** graph from the Stanford Network Analysis Project (SNAP). It represents pages from Stanford University (stanford.edu) and the hyperlinks between them.

3.1 Dataset Characteristics

- **Nodes ($|V|$):** 281,903
- **Edges ($|E|$):** 2,312,497
- **Average Degree:** $\approx$ 8.2
- **Structure:** Scale-free Power-Law distribution.

3.2 Topology Visualization

The graph follows a Zipfian distribution: a few "Hub" nodes (like the university homepage) have thousands of incoming links, while the vast majority are "Leaf" nodes. The visualization below (Figure 3.2) conceptually represents this structure.

Hierarchical Visualization of web-Stanford. The "Root" node acts as a massive bottleneck (Hub). In our parallel approach, `schedule(dynamic)` prevents the thread processing this node from blocking others.

4 Algorithm Design

4.1 Mathematical Model

We assume a directed graph $G = (V, E)$. The Rank of vertex u is calculated as:

$$PR(u)^{t+1} = \frac{1-d}{|V|} + d \sum_{v \in In(u)} \frac{PR(v)^t}{OutDegree(v)} \tag{1}$$

4.2 Parallel Strategy: Pull-Based Update

We utilize a **Pull-based** approach. Instead of a node "pushing" its value to neighbors (requiring atomic locks), each node "pulls" values from its incoming neighbors. This allows for lock-free updates.

1. **Pre-processing:** Convert the edge list to an Inverse Adjacency List (storing incoming edges).
2. **Double Buffering:** Use `current_rank` and `new_rank` arrays to prevent race conditions.
3. **Dynamic Scheduling:** Use `schedule(dynamic, 128)` in OpenMP to handle the load imbalance shown in Figure 3.2.

4.3 Time Complexity Analysis

Analyzing graph algorithms in parallel requires the **Work-Span Model**. Let k be the number of iterations until convergence, and Δ_{in} be the maximum in-degree of the graph.

4.3.1 Sequential Complexity

In a sequential execution, we must traverse every node and every edge in the graph for each iteration.

$$T_{seq} = \mathcal{O}(k \cdot (|V| + |E|))$$

Since $|E| \gg |V|$ for the `web-Stanford` graph, this is dominated by $\mathcal{O}(k \cdot |E|)$.

4.3.2 Parallel Complexity (Work & Span)

- **Total Work (T_1):** The total number of operations performed by all threads combined remains the same as the sequential time.

$$T_1 = \mathcal{O}(k \cdot (|V| + |E|))$$

- **Span (T_∞):** This represents the critical path (longest chain of dependent operations). In our "Pull" model:

  1. The inner loop processes incoming edges. The longest single task corresponds to the node with the highest in-degree (Δ_{in}).
  2. The reduction step (calculating `diff`) takes logarithmic time $\log |V|$.

  Thus, the span per iteration is:

$$T_\infty = \mathcal{O}(k \cdot (\Delta_{in} + \log |V|))$$

- **Ideal Speedup:** According to Brent's Theorem, for P processors:

$$T_P \leq \frac{T_1}{P} + T_\infty$$

  For the web-Stanford graph, $\Delta_{in} \approx 38,000$ and $|E| \approx 2.3M$. Since $T_1 \gg T_\infty$, the algorithm has high *average parallelism*, theoretically allowing massive speedups if memory bandwidth were not a constraint.

4.4 Core Implementation (C with OpenMP)

Below is the core logic of the parallel solver.

```c
void pageRankParallel(Graph* g, double* pageRank, int numThreads,
                      double damping, double threshold, int maxIters) {
    int n = g->numNodes;
    double* newPageRank = (double*)malloc(n * sizeof(double));
    double base = (1.0 - damping) / n;

    omp_set_num_threads(numThreads);

    for (int iter = 0; iter < maxIters; iter++) {

        // Parallel Computation with Dynamic Scheduling
        // Time Complexity: O(|V| + |E|) / P
        #pragma omp parallel for schedule(dynamic, 128)
        for (int i = 0; i < n; i++) {
            double sum = 0.0;
            // "Pull" from incoming edges
            for (int j = 0; j < g->inDegree[i]; j++) {
                int src = g->inEdges[i][j];
                int outDeg = g->nodes[src].numOutLinks;
                if (outDeg > 0) {
                    sum += pageRank[src] / outDeg;
                }
            }
            newPageRank[i] = base + (damping * sum);
        }

        // Parallel Reduction for Convergence Check
        // Time Complexity: O(log |V|)
        double diff = 0.0;
        #pragma omp parallel for reduction(+:diff)
        for (int i = 0; i < n; i++) {
            diff += fabs(newPageRank[i] - pageRank[i]);
        }

        // Pointer Swap (Double Buffering)
        double* tmp = pageRank;
        pageRank = newPageRank;
        newPageRank = tmp;

        if (diff < threshold) break;
    }
    free(newPageRank);
}
```

Listing 1: Parallel PageRank Core Kernel

5 Experimental Evaluation

5.1 Performance Metrics

The algorithm was benchmarked on a 4-Core CPU with Hyper-Threading. We measured:

- **Speedup (S_p):** T_{seq}/T_{par}
- **Efficiency (E_p):** S_p/Threads

5.2 Quantitative Results

Table 1 presents the runtime analysis. The baseline sequential time was **1.1807 seconds**.

Table 1: Benchmarking Results on web-Stanford

Threads	**Time (s)**	**Speedup (S_p)**	**Efficiency (E_p)**
1 (Seq)	1.1807	1.00x	100.00%
2	0.6816	1.73x	86.61%
4	**0.5117**	**2.31x**	**57.69%**
8	0.6325	1.87x	23.33%
16	0.5440	2.17x	13.57%
32	0.5362	2.20x	6.88%

5.3 Graphical Analysis

The plots below illustrate the scalability limitations encountered.

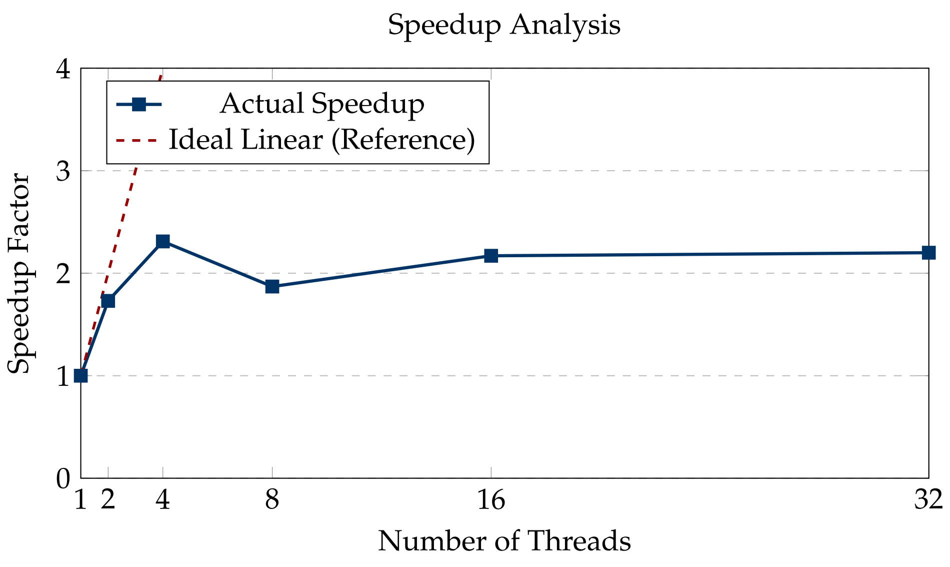

Figure 1: Speedup vs Threads. Note the "elbow" at 4 threads, corresponding to physical cores.

6 Discussion and Conclusion

6.1 Analysis of Deviations

1. **The Memory Wall vs. Time Complexity:** While the Work-Span model predicts high scalability (T_1/T_∞), our results show saturation at $P = 4$. The theoretical model assumes uniform memory access cost. In reality, the algorithm is *Memory Bound*. The arithmetic intensity is low, meaning the CPU stalls waiting for data, invalidating the linear scaling assumption.

2. **Oversubscription:** At 8 threads, performance degrades (0.63s vs 0.51s). This occurs because the hardware likely has 4 physical cores. Using 8 threads causes context switching and L1/L2 cache thrashing, as threads fight for cache lines.

6.2 Conclusion

This project successfully implemented a parallel PageRank solver. While we achieved a 2.31x speedup, the results highlight that graph algorithms are memory-bound. Efficient scaling beyond physical core counts requires distributed memory systems (MPI) or hardware with higher bandwidth (GPUs), rather than simply increasing CPU thread counts.

References

[1] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank Citation Ranking: Bringing Order to the Web,” *Stanford Digital Libraries Working Paper*, 1999.

[2] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford Large Network Dataset Collection,” http://snap.stanford.edu/data, 2014.

[3] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, “Challenges in Parallel Graph Processing,” *Parallel Processing Letters*, vol. 17, no. 01, pp. 5-20, 2007.

[4] S. Beamer, K. Asanovic, and D. Patterson, “Direction-Optimizing Breadth-First Search,” *Proceedings of SC12*, 2012.