ALGORITHM DESIGN DOCUMENT

Parallel PageRank Engine (OpenMP)

Course: CS-478 — Design and Analysis of Algorithms

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

December 24, 2025

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to detail the architectural design, data structures, and algorithmic strategies used to implement the Parallel PageRank solver. This system is designed to process large-scale graph datasets (e.g., web-Stanford) efficiently using Shared Memory Parallelism.

1.2 Scope

The system accepts a graph edge list as input, constructs an optimized in-memory representation, and utilizes the OpenMP framework to calculate the PageRank vector. The scope is limited to:

- Single-node, multi-core execution (Shared Memory).
- Static graph analysis (Graph does not change during computation).
- Pull-based iterative convergence.

2 System Architecture

2.1 High-Level Data Flow

The system follows a standard Extract-Load-Transform (ELT) pattern adapted for graph processing.

System Architecture Diagram. The Graph Loader acts as a pre-processor, converting raw edges into an optimized "Inverse Adjacency" format required by the Parallel Engine.

2.2 Design Strategy: Pull-Based Execution

To maximize parallelism and avoid the overhead of atomic locks (mutexes), we adopted a **Pull-Based** design:

- **Push (Classic):** Node u adds its rank to neighbor v. Requires locking v.
- **Pull (Selected):** Node v reads ranks from all incoming neighbors $u \in In(v)$. No locks required as only thread v writes to v's new rank.

3 Data Structure Design

The performance of graph algorithms is dictated by memory layout. We use a modified Adjacency List optimized for the Pull method.

3.1 The Graph Structure

```
typedef struct {
    int* outLinks;       // Array of outgoing neighbor IDs
    int numOutLinks;     // Count (Out-Degree)
    int capacity;        // For dynamic resizing during load
} Node;

typedef struct {
    int numNodes;        // Total Vertices |V|
    long numEdges;       // Total Edges |E|
```

```
    Node* nodes;         // Array of Node objects

    // CRITICAL FOR PARALLEL PERFORMANCE
    int** inEdges;       // Array of arrays (Inverse Adjacency List)
    int* inDegree;       // Count of incoming edges
} Graph;
```

Listing 1: Core Data Structures

3.2 Memory Justification

- **inEdges (‘int**‘):** Explicitly storing incoming edges doubles memory usage ($2 \cdot |E|$) but allows $O(1)$ access to a node's dependencies during the parallel "Pull" phase.
- **outLinks:** Stored only to calculate $OutDegree(u)$ during the update phase.
- **Arrays vs Linked Lists:** We use contiguous arrays (‘malloc‘) instead of linked lists to improve CPU Cache Locality (Spatial Locality).

4 Algorithm Design

4.1 Execution Flow

The algorithm operates in synchronous "Supersteps" (Bulk Synchronous Parallel - BSP).

1. **Initialization:**
   - Allocate ‘current$_r ank[]$‘and‘$new_r ank[]$‘.
   - Set all ranks to $1/N$.
2. **Iterative Step (Parallel Region):**
   - **Work Distribution:** Use OpenMP ‘schedule(dynamic, 128)‘ to assign nodes to threads.
   - **Computation:** For each node v, iterate over ‘inEdges[v]‘, sum probabilities, apply damping factor.
   - **Global Barrier:** Wait for all threads to finish computing.
3. **Convergence Check:**
   - Calculate L_1 norm (Manhattan distance) between ‘current‘ and ‘new‘.
   - Perform Parallel Reduction on the difference variable.
4. **Pointer Swap:** Swap ‘current‘ and ‘new‘ pointers (Double Buffering) to avoid ‘memcpy‘ overhead.

4.2 Convergence Logic

$$\text{Diff} = \sum_{v \in V} |PR_{new}(v) - PR_{old}(v)|$$

The algorithm terminates when Diff $< \epsilon$ (Threshold 10^{-6}).

5 Interface Specifications

The system is modularized into three primary components: Graph Management, Solver, and Benchmarking.

5.1 Graph Management Module

Handles memory allocation and file I/O.

- `Graph* createGraph(int numNodes)`: Allocates zero-initialized memory.
- `void addEdge(Graph* g, int src, int dst)`: Thread-unsafe; must be called sequentially during loading. Handles array resizing.
- `Graph* loadGraphFromFile(const char* filename)`: Wrapper for parsing SNAP datasets.

5.2 PageRank Solver Module

The core computational kernel.

```
void pageRankParallel(
    Graph* g,           // The Data Structure
    double* pageRank,   // Output Buffer
    int numThreads,     // Parallelism Factor
    double damping,     // Alpha (0.85)
    double threshold,   // Epsilon (1e-6)
    int maxIters        // Safety cap
);
```

5.3 Parallel Directives Used

- `#pragma omp parallel for`: Main parallelization construct.
- `reduction(+:diff)`: For aggregating the convergence error.
- `schedule(dynamic)`: To handle the Power-Law distribution of the SNAP dataset.

6 System Requirements

6.1 Hardware

- **CPU:** Multi-core processor (Intel i5/i7 or AMD Ryzen).
- **RAM:** Minimum 4GB (Required to store the expanded ‘web-Stanford‘ graph structure).

6.2 Software

- **OS:** Linux (Ubuntu 20.04+) or Windows (via WSL).
- **Compiler:** GCC (g++) with OpenMP support (libgomp).
- **Standard:** C99 or C11.