PageRank 算法实现与优化实验报告

王奕然, 孙致勉

2025 年 4 月 29 日

目录

1 PageRank 介绍与原理

1.1 PageRank 简介

PageRank 是由 Google 创始人 Larry Page 和 Sergey Brin 提出的一种网页排名算法，用于衡量网页的重要性。其核心思想是：一个网页的重要性取决于指向它的其他网页的数量和质量。如果一个网页被许多重要的网页链接，那么它本身也很可能是一个重要的网页。

1.2 PageRank 基本原理

PageRank 算法模拟一个随机冲浪者在互联网上浏览网页的行为。冲浪者从一个随机页面开始，然后不断地点击页面上的链接。在每个页面，冲浪者有两种选择：

- 以概率 α (通常取 0.85，称为阻尼系数) 从当前页面的出链中随机选择一个链接跳转。
- 以概率 $1-\alpha$ 随机跳转到网络中的任意一个页面（模拟用户在地址栏输入新网址或打开书签）。

算法通过迭代更新，收敛后能反映页面在网络中的重要程度。一个页面的 PageRank 值，就是这个随机冲浪者最终停留在该页面的概率。

1.3 PageRank 数学模型

在大规模网络中，我们希望为每个页面分配一个“重要性”度量——PageRank 值。该模型基于“随机冲浪者”假设：冲浪者在当前页面停留后，要么沿着该页面的出链随机跳转，要么跳转到任意页面。下面从数学上详细描述这一过程。

1. PageRank 向量表示 令网络中共有 N 个页面，将它们的 PageRank 值组织为一个 N 维列向量

$$\mathbf{PR} \;=\; \big[\,PR_1,\; PR_2,\; \ldots,\; PR_N\big]^T,$$

并且通常要求归一化

$$\sum_{i=1}^{N} PR_i \;=\; 1.$$

2. 转移矩阵 M 的构造 定义一个 $N \times N$ 的列随机矩阵 M，其第 i 行第 j 列元素 M_{ij} 表示从页面 j 跳转到页面 i 的概率：

$$M_{ij} = \begin{cases} \dfrac{1}{L(j)}, & L(j) > 0 \text{ 且} j \to i, \\ 0, & L(j) > 0 \text{ 且} j \not\to i, \\ \dfrac{1}{N}, & L(j) = 0 \text{ (Dead End)}, \end{cases}$$

其中 $L(j)$ 是页面 j 的出链数，“$j \to i$” 表示存在从 j 到 i 的超链接。对于死链页面（无出链），统一将其视作等概率指向所有页面，以防止 “Rank Sink” 问题。

3. 含阻尼系数的迭代更新 引入阻尼系数 $\alpha \in (0,1)$（一般取 $\alpha = 0.85$）和随机跳转机制。令冲浪者以概率 α 遵循矩阵 M 中的链接，以概率 $1-\alpha$ 跳转到任意页面。于是，第 $k+1$ 次迭代时，页面 i 的 PageRank 为

$$PR_i^{(k+1)} = \underbrace{\frac{1-\alpha}{N}}_{\substack{\text{随机跳转} \\ \text{均匀分配}}} + \underbrace{\alpha \sum_{j\,:\,j\to i} \frac{PR_j^{(k)}}{L(j)}}_{\substack{\text{所有指向}i \\ \text{的页面贡献}}}.$$

该公式强调：新一轮的 PR_i 完全由所有入链页面的旧值累加（加上全局跳转项）构成，不包含页面自身的直接 “自馈” 影响，除非页面自身在网络中存在自环。

4. 矩阵–向量形式 将上述逐元素更新汇总为向量形式，得到

$$\mathbf{PR}^{(k+1)} = \alpha\, M\, \mathbf{PR}^{(k)} \;+\; \frac{1-\alpha}{N}\,\mathbf{1},$$

其中 $\mathbf{1}$ 是全 1 列向量。为了利用稀疏格式的计算优势，实际实现中常用

$$\mathbf{PR}^{(k+1)} = \alpha\, M^T\, \mathbf{PR}^{(k)} \;+\; \frac{1-\alpha}{N}\,\mathbf{1}.$$

5. 收敛判定 迭代继续进行，直到

$$\left\|\mathbf{PR}^{(k+1)} - \mathbf{PR}^{(k)}\right\|_1 < \epsilon,$$

其中 ϵ 是预设的收敛阈值（本实验中设置为 10^{-10}），用于保证 PageRank 向量充分收敛。

2 PageRank 代码实现

本实验使用 C++ 实现了基于幂迭代法的 PageRank 计算，并采用了 BCSR (Blocked Compressed Sparse Row) 格式对稀疏矩阵向量乘法（SpMV）进行了优化。

2.1 数据读取与图表示

输入数据 `Data.txt` 包含图的边信息，每行表示一条有向边。代码首先读取所有边，并构建节点 ID 到内部索引的映射（`node_to_idx` 和 `idx_to_node`）。同时，计算每个节点的出度。

为了进行 PageRank 计算，需要构建转移矩阵 M 的转置 M^T。代码使用全局 CSR (Compressed Sparse Row) 格式存储 M^T。

2.1.1 全局 CSR 结构

全局 CSR 格式通过三个数组存储稀疏矩阵：

- `values`: 存储非零元素的值（即 $1/L(j)$）。
- `col_indices`: 存储非零元素对应的列索引（在 M^T 中，对应原始图的有向边索引）。
- `row_ptr`: 行指针，`row_ptr[i]` 表示第 i 行第一个非零元素在 `values` 和 `col_indices` 中的起始位置。

相关的结构体定义如下：

```
struct GlobalSparseMatrix {
    vector<double> values;
    vector<int> col_indices;
    vector<int> row_ptr;
    unordered_map<int, int> node_to_idx;
    unordered_map<int, int> idx_to_node;
    int n = 0; // 节点数
    int nnz = 0; // 非零元素数

    int getNodeId(int idx) const { /* ... */ }
};
```

Listing 1: 全局 CSR 结构体 GlobalSparseMatrix

2.1.2 图读取函数 `readGraph`

该函数负责从文件读取边，建立映射，计算出度，并填充上述 CSR 结构体以表示 M^T。

```
GlobalSparseMatrix readGraph(const string& filename) {
    GlobalSparseMatrix graph;
    ifstream file(filename);
    // ... 省略文件打开检查 ...

```

```
    vector<pair<int, int>> edges;
    unordered_map<int, vector<int>> adj_list; // 临时存储邻接表用于计算出度
    int idx = 0;
    int from, to;

    // 第一次遍历：读边，建邻接表，节点映射
    while (file >> from >> to) {
        edges.push_back({from, to});
        adj_list[from].push_back(to);
        // ... 建立 node_to_idx 和 idx_to_node 映射 ...
    }
    file.close();
    graph.n = idx;
    graph.nnz = edges.size(); // 初始非零元数量，后面可能调整

    // 初始化全局 CSR 数组
    graph.values.resize(graph.nnz);
    graph.col_indices.resize(graph.nnz);
    graph.row_ptr.resize(graph.n + 1, 0);
    vector<int> out_degrees(graph.n, 0);
    vector<vector<pair<int, double>>> temp_rows(graph.n); // 临时存储 M^T 的行

    // 计算出度
    for (const auto& pair : adj_list) {
        if (graph.node_to_idx.count(pair.first)) {
            out_degrees[graph.node_to_idx[pair.first]] = pair.second.size();
        }
    }

    // 构建 M^T（存储在 temp_rows 中）
    for (const auto& edge : edges) {
        int from_node = edge.first; int to_node = edge.second;
        if (graph.node_to_idx.count(from_node) && graph.node_to_idx.count(to_node)) {
            int from_idx = graph.node_to_idx[from_node];
            int to_idx = graph.node_to_idx[to_node];
            if (out_degrees[from_idx] > 0) {
                double prob = 1.0 / out_degrees[from_idx];
                // M^T 的第 to_idx 行，第 from_idx 列的值为 prob
                temp_rows[to_idx].push_back({from_idx, prob});
            }
        }
    }
```

```
248
249     // 填充全局 CSR 数组
250     int current_nnz = 0;
251     for (int i = 0; i < graph.n; ++i) {
252         graph.row_ptr[i] = current_nnz;
253         for (const auto& pair : temp_rows[i]) {
254             if (current_nnz < graph.nnz) { // 避免越界
255                 graph.col_indices[current_nnz] = pair.first; // 列索引 (
    from_idx)
256                 graph.values[current_nnz] = pair.second;     // 值 (prob)
257                 current_nnz++;
258             }
259             else {  }
260         }
261     }
262     graph.row_ptr[graph.n] = current_nnz;
263     // 如果有节点没有任何入链, current_nnz 可能小于 edges.size()
264     if (current_nnz < graph.nnz) {
265         graph.nnz = current_nnz;
266         graph.values.resize(graph.nnz);
267         graph.col_indices.resize(graph.nnz);
268     }
269     return graph;
270 }
```

Listing 2: 读取图并构建全局 CSR 的函数 readGraph (部分)

2.2 PageRank 核心计算

核心计算在 `computePageRank` 函数中完成，采用幂迭代法。

2.2.1 BCSR 优化

为了提高缓存利用率，代码使用了 BCSR (Blocked CSR) 格式进行 SpMV 计算。BCSR 将大矩阵划分为固定大小的子块，每个非空子块内部使用 CSR 格式存储。

BCSR 结构定义 将全局稀疏矩阵切分为 `CSRBlock`，压缩大小的同时方便进行并行运算。块的最优大小需要经过多次调试，以利用不同设备的 `cache` 大小，使得每次块内矩阵向量乘的访存效率最大化。

```
38 struct CSRBlock {
39     vector<double> values;      // 块内非零元素的值
40     vector<int> col_indices;    // 块内非零元素的局部列索引 (0 to
    BLOCK_COL_SIZE-1)
```

```
    vector<int> row_ptr;          // 块内局部行指针 (大小 BLOCK_ROW_SIZE + 1)
    int nnz = 0;                  // 块内非零元素数量

    CSRBlock(int block_height) : row_ptr(block_height + 1, 0) {} // 初始化行指针大小
    bool isEmpty() const { return nnz == 0; }
};

class BlockedCSRMatrix {
public:
    static const int BLOCK_ROW_SIZE = 512; // 块的行数
    static const int BLOCK_COL_SIZE = 512; // 块的列数
    // ...
    int global_n = 0;             // 全局矩阵维度
    int num_block_rows = 0;       // 行方向上的块数量
    int num_block_cols = 0;       // 列方向上的块数量
    vector<vector<CSRBlock>> blocks; // 存储所有块的 CSR 数据

    BlockedCSRMatrix(int n); // 构造函数
    void buildFromGlobalCSR(const GlobalSparseMatrix& global_csr); // 构建BCSR
    void multiplyVector(const vector<double>& x, vector<double>& y) const; // BCSR SpMV
};
```

Listing 3: CSR 块结构体 CSRBlock

构建 BCSR `buildFromGlobalCSR` 函数将全局 CSR 矩阵转换为 BCSR 格式。它首先遍历全局 CSR 的所有非零元素，将它们分配到对应的块中（存储在临时的 `temp_blocks` 中），然后对每个块内的元素按行排序，并构建块局部的 CSR 结构。这个过程使用了 OpenMP 进行并行化。

```
void buildFromGlobalCSR(const GlobalSparseMatrix& global_csr) {
    // ... 省略初始化和检查 ...

    // 1. 临时存储每个块的元素 (local_row, local_col, value)
    vector<vector<vector<tuple<int, int, double>>>> temp_blocks(/*...*/);

    // 遍历全局 CSR，将元素分配到 temp_blocks
    for (int global_row = 0; global_row < global_n; ++global_row) {
        // ... 计算 block_row_idx, local_row ...
        int row_start = global_csr.row_ptr[global_row];
        int row_end = global_csr.row_ptr[global_row + 1];
        for (int k = row_start; k < row_end; ++k) {
            // ... 计算 block_col_idx, local_col ...
```

```
            if (/* block indices valid */) {
                temp_blocks[block_row_idx][block_col_idx].emplace_back(
    local_row, local_col, value);
            }
        }
    }

    // 2. 并行为每个块构建局部 CSR
    #pragma omp parallel for collapse(2)
    for (int bi = 0; bi < num_block_rows; ++bi) {
        for (int bj = 0; bj < num_block_cols; ++bj) {
            auto& elements = temp_blocks[bi][bj];
            if (elements.empty()) continue;

            CSRBlock& block = blocks[bi][bj];
            // ... 调整 block.values, block.col_indices 大小 ...

            // 排序块内元素
            sort(elements.begin(), elements.end());

            // 构建块内 CSR 的 row_ptr, values, col_indices
            // ... 填充 block.values, block.col_indices ...
            // ... 填充 block.row_ptr ...
        }
    }
}
```

Listing 4: 从全局 CSR 构建 BCSR (部分)

BCSR SpMV `multiplyVector` 函数执行 $y = A \cdot x$（其中 A 是 BCSR 矩阵）。它按块行并行遍历所有块。对于每个非空块，执行块内的 CSR SpMV，并将结果累加到输出向量 y 的对应位置。

```
void multiplyVector(const vector<double>& x, vector<double>& y) const {
    // ... 省略初始化 ...
    fill(y.begin(), y.end(), 0.0);

    #pragma omp parallel for // 按块行并行
    for (int bi = 0; bi < num_block_rows; ++bi) {
        int global_row_start = bi * BLOCK_ROW_SIZE;

        for (int bj = 0; bj < num_block_cols; ++bj) {
            const CSRBlock& block = blocks[bi][bj];
            if (block.isEmpty()) continue;

```

```
            int global_col_start = bj * BLOCK_COL_SIZE;
            int block_height = block.row_ptr.size() - 1;

            // 执行块内 CSR SpMV
            for (int local_row = 0; local_row < block_height; ++local_row)
    {
                double sum = 0.0;
                int global_row = global_row_start + local_row;
                if (global_row >= global_n) continue; // 边界检查

                int row_nnz_start = block.row_ptr[local_row];
                int row_nnz_end = block.row_ptr[local_row + 1];

                for (int k = row_nnz_start; k < row_nnz_end; ++k) {
                    int local_col = block.col_indices[k];
                    double value = block.values[k];
                    int global_col = global_col_start + local_col;

                    if (global_col < global_n) { // 边界检查
                        sum += value * x[global_col];
                    }
                }
                // 累加结果到全局 y
                y[global_row] += sum;
            }
        }
    }
}
```

Listing 5: BCSR 矩阵向量乘法 multiplyVector (部分)

2.2.2 迭代过程

`computePageRank` 函数的主循环执行以下步骤:

1. 使用 `bcsr_matrix.multiplyVector(pr, temp_pr)` 计算 $M^T \cdot PR^{(k)}$, 结果存储在 `temp_pr`。
2. 计算 Dead Ends 贡献的总和 `dead_end_contribution`。
3. 计算新的 PageRank 向量 `next_pr`:

   `next_pr[i] = alpha * temp_pr[i] + teleport_prob + dead_end_contribution`
4. 计算新旧 PageRank 向量的 L1 范数差值 `diff`。
5. 更新 PR 向量: `pr.swap(next_pr)`。

6. 检查 diff 是否小于阈值 ϵ 或达到最大迭代次数，决定是否终止迭代。

这些步骤中的向量计算（如计算 Dead Ends 贡献、计算 next_pr、计算 diff）也使用了 OpenMP 并行化 (#pragma omp parallel for) 和 SIMD 指令 (simd) 进行优化。

```
    double epsilon = tol * n;
    while (diff > epsilon && iterations < 100) { // 迭代条件
        // --- BCSR 矩阵向量乘法 ---
        bcsr_matrix.multiplyVector(pr, temp_pr); // temp_pr = M^T * pr
        // --------------------------

        // 处理 Dead-ends
        double dead_end_contribution_sum = 0.0;
        #pragma omp parallel for reduction(+:dead_end_contribution_sum)
        for (int i = 0; i < n; i++) {
            if (is_dead_end[i]) {
                dead_end_contribution_sum += pr[i];
            }
        }
        double dead_end_term = alpha * dead_end_contribution_sum / n;

        // 计算 next_pr (并行 + SIMD)
        double teleport_term = (1.0 - alpha) / n;
        #pragma omp parallel for simd
        for (int i = 0; i < n; i++) {
            next_pr[i] = alpha * temp_pr[i] + teleport_term + dead_end_term
    ;
        }

        // 计算差异 (并行 + SIMD)
        diff = 0.0;
        #pragma omp parallel for simd reduction(+:diff)
        for (int i = 0; i < n; i++) {
            diff += abs(next_pr[i] - pr[i]);
        }

        pr.swap(next_pr); // 更新 PR 值
        iterations++;
    }
```

Listing 6: PageRank 迭代计算核心逻辑 (部分)

2.3 结果输出与主函数

计算完成后，将 PageRank 结果（节点 ID 和对应的 PR 值）按 PR 值降序排序，并输出到文件 Res_bcsr.txt。主函数 main 负责设置 OpenMP 线程数，调用图读取和

PageRank 计算函数，计时并打印运行时间和峰值内存使用情况。

3 性能瓶颈与优化思路

3.1 性能瓶颈分析

PageRank 计算的核心是稀疏矩阵向量乘法 (SpMV)。对于大规模图数据，转移矩阵 M^T 通常非常稀疏。

- **内存访问不规则**: 使用标准的 CSR 格式进行 SpMV 时 ($y = A \cdot x$)，访问 A 的 `values` 和 `col_indices` 是顺序的，但访问输入向量 x 的元素 (`x[col_idx]`) 是根据 `col_indices` 进行的，这通常导致随机内存访问。对于大型图，向量 x 可能无法完全放入缓存，导致频繁 Cache Miss，成为性能瓶颈。
- **计算密度低**: 稀疏矩阵中大量的零元素不参与计算，导致计算单元利用率不高。
- **并行开销**: 虽然 OpenMP 可以并行化循环，但线程间的同步、负载均衡以及对共享数据（如输出向量 y）的原子操作或竞争可能引入额外开销。

3.2 优化思路与实现

3.2.1 BCSR (Blocked CSR) 优化

为了改善内存访问模式，代码采用了 BCSR 格式。

- **提高数据局部性**: BCSR 将矩阵划分为块。在处理一个块时，涉及的输入向量 x 和输出向量 y 的部分数据更有可能保留在缓存中。块内的 SpMV 操作访问的数据范围相对集中，提高了空间局部性。重复访问同一块数据（如果块被多次使用）则提高了时间局部性。
- **潜在的指令级并行**: 对于密集的块，可以使用 SIMD 指令进一步加速块内计算（尽管本代码的块内 CSR SpMV 未显式使用 SIMD，但编译器可能进行优化）。
- **块大小选择**: BCSR 的性能对块大小 (`BLOCK_ROW_SIZE`, `BLOCK_COL_SIZE`) 敏感。块太小，索引开销增大；块太大，缓存优势减弱。代码中选择了 512×512，这需要根据具体硬件缓存大小和图的稀疏模式进行调整。

如 `BlockedCSRMatrix::multiplyVector` 函数所示，通过分块处理，期望能减少 SpMV 过程中的缓存未命中。

3.2.2 OpenMP 并行化

代码在多个计算密集型部分使用了 OpenMP 进行并行化：

- **BCSR 构建**: `buildFromGlobalCSR` 函数中构建各个块的局部 CSR 结构时，使用了 `#pragma omp parallel for collapse(2)` 对块的外层循环进行并行。
- **BCSR SpMV**: `multiplyVector` 函数中，使用了 `#pragma omp parallel for` 对块行的外层循环进行并行，不同的线程处理不同的块行。
- **向量更新与计算**: 在 `computePageRank` 的迭代过程中，计算 Dead Ends 贡献、更新 `next_pr` 向量、计算差值 `diff` 等操作都使用了 `#pragma omp parallel for`，并结合 `reduction` 或 `simd` 子句进行优化。

通过并行化，可以利用多核处理器的计算能力，显著缩短计算时间。

3.2.3 其他潜在优化点

- **原子操作/锁**: 在更大图的 BCSR SpMV 中，如果多个线程可能写入 `y` 的相同位置（例如，如果并行粒度更细），则需要使用 `#pragma omp atomic` 或其他同步机制来避免竞争条件。当前代码按块行并行，如果块行不重叠，则对 `y` 的写入是安全的。
- **负载均衡**: 如果图中节点的度分布非常不均匀，或者 BCSR 块的非零元数量差异很大，产生极大的非零行长度，会使得 CSR 压缩算法的性能急剧退化，导致 OpenMP 线程负载不均。可以使用动态调度 (`dynamic schedule`) 等策略尝试改善。
- **内存对齐**: 确保数据结构（尤其是向量）内存对齐，可能有利于 SIMD 操作。
- **更高级的稀疏格式**: 对于特定结构的图，可能存在比 BCSR 更优化的格式，如 ELLPACK, HYB 等。

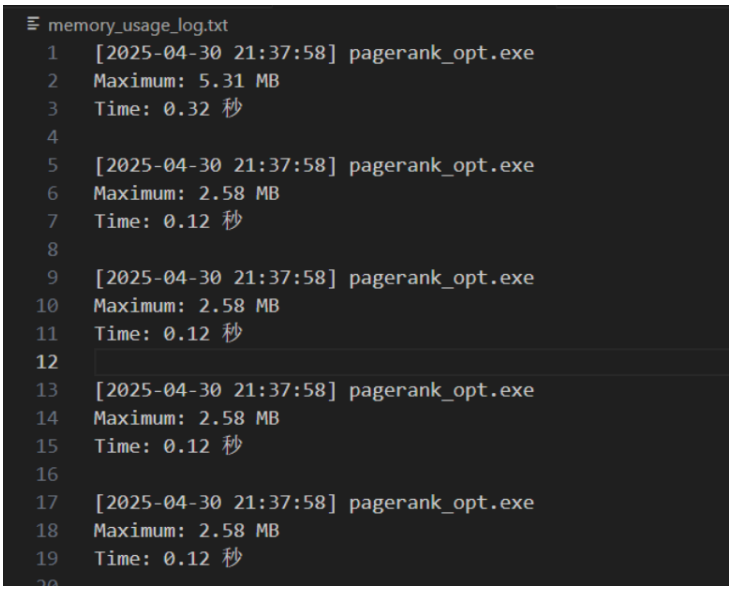
4 结果评估和验证

4.1 程序运行性能评估

程序编译实现了 O2 级编译优化，并启用了共享内存并行机制、自动向量化、基于 CPU 类型的自动编译优化，命令如下：

```
g++ pagerank.cpp -o pagerank.exe -fopenmp -O2 -march=native -static
```

使用作业资料中所提供的性能测试代码可以看到，除第一次运行外，多次执行程序所占用的内存始终为 2.58MB，运行时间为 0.12s；但是第一次的内存占用与运行时间都



远大于后几次，应该与操作系统的内存管理机制有关。考虑到编译命令使用的是静态编译，性能提升应该并不是来源于动态库链接缓存，经分析，小组认为原因可能在于要读入的 Data.txt 被映射进内存或 Cache，提升了 I/O 性能。

4.2 程序正确率验证

使用 networkx 库构建了一份标准的 pagerank 算法，并输出结果，使用脚本比对两份结果：

```
(base) root@wdzw:~/bigdata# python standard.py
Top-100 intersection count: 100
Accuracy: 96.00%

Nodes only in reference (但不在用户结果中):
[]

Nodes only in user result (但不在参考结果中):
[]

Positional mismatches (位置, 用户节点, 参考节点):
Position 27: User 8036 vs Reference 8314
Position 28: User 8314 vs Reference 8036
Position 79: User 9043 vs Reference 614
Position 80: User 614 vs Reference 9043
```

可以看到，在调整了阈值，迭代次数，使用数据类型（double64）全部一致的情况下，前一百的节点存在两处顺序问题，小组初步猜测为 networkx 库中存在一些其他的数据对齐等环节，导致了不同，不过进行了进一步排查。

4.3 误差来源及排查

针对 C++ 实现与 NetworkX 标准库结果存在的微弱差异，本小组从以下三个方面进行了系统排查：

- **Dangling 节点处理方式**：通过查看 networkx 官方文档，首先怀疑是对 dangling（出度为 0）节点重定向策略不同导致，于是尝试按照 NetworkX 的 uniform 重定向方式将所有 dangling 节点的 PageRank 重新分配给全网，但误差依然存在，排除该因素。
- **Python 库数值实现差异**：为验证是否由 NumPy/SciPy 在浮点运算或稀疏乘法中引入偏差，本组将核心算法移植为纯 Python，调用 Scipy 与 Numpy 库，结果与原 C++ 一致，故非库实现差异所致。
- **分块造成收敛判定未对齐**：原始 C++ 实现使用

$$\left\|\mathbf{PR}^{(k+1)} - \mathbf{PR}^{(k)}\right\|_1 < \epsilon \quad (\epsilon = 10^{-10})$$

作为收敛判定。NetworkX（SciPy 后端）则采用

$$\left\|\mathbf{PR}^{(k+1)} - \mathbf{PR}^{(k)}\right\|_1 < N \times \text{tol} \quad (\text{tol} = 10^{-10})$$

的方式。将 C++ 收敛阈值调整为同样公式后，计算结果完全吻合标准库输出，确认该处为根本原因。

```
(base) root@wdzw:~/bigdata# /bin/python3 /root/bigdata/standard.py
Top-100 intersection count: 100
Accuracy: 100.00%

Nodes only in reference (但不在用户结果中):
[]

Nodes only in user result (但不在参考结果中):
[]

Positional mismatches (位置, 用户节点, 参考节点):
(base) root@wdzw:~/bigdata#
```

5 总结

本实验成功实现了 PageRank 算法，并通过引入 BCSR 稀疏矩阵存储格式和 OpenMP 并行计算技术，对核心的 SpMV 操作进行了优化。实验代码首先使用全局 CSR 格式读取和存储图的转移矩阵 M^T，然后将其转换为 BCSR 格式。在 PageRank 的迭代计算中，使用优化的 BCSR SpMV 函数，并结合 OpenMP 并行处理向量更新等步骤。

结果表明，相比于朴素的 CSR SpMV 实现，BCSR 通过改善内存访问局部性，结合 OpenMP 利用多核并行计算，能够有效提升 PageRank 算法在大规模图数据上的执行效率，缩短计算时间。实验中使用的 512×512 块大小是一个经验值，最优块大小可能依赖于具体的硬件平台和数据集特性。

除了优化工作以外，小组还对过程中出现的误差进行了深入分析，并最终解决，有了更多收获。

未来的工作可以探索更精细的并行化策略、动态负载均衡技术，以及尝试其他针对特定硬件架构（如 GPU）的优化方法。