[redacted]

课 程 实 验 报 告

课程名称：大数据系统综合实践

专业班级：[redacted]

学　　号：[redacted]

姓　　名：[redacted]

指导教师：[redacted]

报告日期：[redacted]

计算机科学与技术学院

目 录

实验要求 1

实验过程 2

2.1 社区发现算法 2

2.1.1 GN 算法 2

2.1.2 Louvain 算法 2

2.1.2.1 模块度(modularity) 2

2.1.2.2 Louvain 算法步骤 3

2.1.2.3 模块度增益(modularity gain) 4

2.1.2.4 对无向无权图的处理 4

2.2 环境选择及搭建 4

2.2.1 框架与编程语言 5

2.2.2 脚本工具 5

2.2.3 profiling 工具选择 5

2.3 优化一：IO 优化 6

2.4 优化二：parse 优化（系统调用与 SIMD 优化） 6

2.5 优化三：并行优化 7

2.5.1 并行优化解析数据 7

2.5.2 并行优化构建邻接表 8

2.6 优化四：减少重复计算 8

实验结果 9

实验总结 11

实验要求

本实验旨在通过大规模图数据中社区发现算法的设计与性能优化，帮助学生深入理解图计算系统的工作原理和性能优化机制，并学会使用图计算框架进行大规模图数据分析和处理。通过此实验，学生将能够掌握图计算的基本原理，编写比较复杂的图算法程序并进行性能调优。

- 操作系统：Linux
- 编译器：gcc 或者 g++ 4.8 以上
- Make：GNU make 4.0 以上
- 框架：Spark GraphX、Pregel、或你所熟悉擅长的其它框架

数据集	说明
soc-LiveJournal1	顶点数：4.8 million、边数：69 million 来源：https://snap.stanford.edu/data/soc- LiveJournal1.html
cit-HepPh	顶点数：34546, 边数：421578 来源：https://snap.stanford.edu/data/cit-HepPh.htm

本实验有功能和性能两方面的要求，首先需要能够正确地统计出所给数据集中的所有社区，其次需要不断优化你所设计的算法（并行优化、存储优化、通信优化等），使得在最终的测试机器上执行时间最短，最终的成绩和你的算法优化效果密切相关。

实验过程

代码详见 GitHub 仓库 https://github.com/vaaandark/louvain-rs

2.1 社区发现算法

社区划分问题主要有两种思路：

1. 凝聚方法(agglomerative method)：添加边
2. 分裂方法(divisive method)：移除边
3. 相似性检测方法：同一个社区内的节点，之所以能够聚集在一起，是因为它们有相似性。因此只要能够将一个节点很好地表示，成为一个向量，那么同样可以用相似性大法来寻找社区聚集，不过这点上需要向量对节点的描述足够好和足够完备。

2.1.1 GN 算法

GN 算法的核心思想是通过移除网络中的边来构造社区结构。它认为社区之间的连接较少，而社区内部的连接较多。GN 算法会逐步移除网络中具有最高“节点介数中心性”(Betweenness Centrality)的边，直到网络被分割成若干个社区。其算法步骤如下：

1. 初始化：将整个网络视为一个单一的社区。
2. 计算中介中心性：对于网络中的每一条边，计算其 Betweenness Centrality。
3. 移除边：移除中介中心性最高的边。
4. 重复：重复上述步骤，直到网络被分割成预定的社区数量或满足其他停止条件。

GN 算法的时间复杂度高，删除了边之后需要重新计算边介数，终止条件也不好确定。即使引入了模块度(modularity)，并使用了自底向上方法合并节点计算，也只能解决终止条件的问题，并不能解决计算复杂度高的问题。因此，迫切地需要一种更快更好的算法！

2.1.2 Louvain 算法

2.1.2.1 模块度(modularity)

模块度(modularity)：模块度是一种衡量网络社区结构强度的方法，它通过比较社区内边的权重和与随机情况下社区内边的权重和的差异来评估社区划分的质量。模块度的值越接近 1，表示社区划分的效果越好，即社区内部的连接越紧密。它的定义如下：

$$Q = \frac{1}{2m} \sum_{i,j} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j)$$

$$\delta(u, v) = \begin{cases} 1 \text{ when } u = v \\ 0 \text{ else} \end{cases}$$

其中：

- A_{ij} 表示节点 i 和节点 j 之间边的权重
- $k_1 = \sum_j A_{ij}$
- c_i 表示节点 i 所属的社区
- $m = \frac{1}{2} \sum_{ij} A_{ij}$ 表示所有边的权重和

经过化简后有：

$$Q = \frac{1}{2m} \sum_{c} \left[\sum \text{in} - \frac{\left(\sum \text{tot} \right)^2}{2m} \right]$$

其中 $\sum \text{in}$ 表示社区 c 里面边的权重之和，$\sum \text{tot}$ 表示与社区 c 内节点相连的边的权重之和。

进一步简化有：

$$\begin{aligned} Q &= \sum_{c} \left[\frac{\sum \text{in}}{2m} - \left(\frac{\sum \text{tot}}{2m} \right)^2 \right] \\ &= \sum_{c} [e_c - a_c^2] \end{aligned}$$

2.1.2.2 Louvain 算法步骤

1. 将图中的每个节点看成一个独立的社区。
2. 对每个节点 i，依次尝试把节点 i 分配到其每个邻居节点所在的社区，计算分配前与分配后的模块度变化 ΔQ，并记录 ΔQ 最大的那个邻居节点，如果 $\max \Delta Q > 0$，则把节点 i 分配 ΔQ 最大的那个邻居节点所在的社区，否则保持不变。
3. 重复 2.，直到所有节点的所属社区不再变化。

4. 对图进行压缩, 将所有在同一个社区的节点压缩成一个新节点, 社区内节点之间的边的权重转化为新节点的环的权重，社区间的边权重转化为新节点间的边权重。
5. 重新回到 1. 开始计算，直到整个图的模块度不再发生变化。

2.1.2.3 模块度增益(modularity gain)

Louvain 相对 GN 算法的最大优势就是计算复杂度小，很多东西不需要重复计算，尤其是对每个点计算它移动到相邻社区获得的模块度增益。

模块度增益看似需要把移动前和移动后的模块度完整地计算出来再相减，但是并不需要，可以对公式进行化简：

$$\begin{aligned}\Delta Q &= \left[\frac{\sum \mathrm{in} + k_{i,j}}{2m} - \left(\frac{\sum \mathrm{tot} + k_i}{2m}\right)^2\right] - \left[\frac{\sum \mathrm{in}}{2m} - \left(\frac{\sum \mathrm{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right] \\ &= \left[\frac{k_{i,\mathrm{in}}}{2m} - \frac{\sum \mathrm{tot}\ k_i}{2m^2}\right]\end{aligned}$$

其中：

- $k_{i,\mathrm{in}}$ 是节点 i 到社区的边的权重和
- $\sum \mathrm{tot}$ 是社区所有点的边的权重和之和
- k_i 是节点的所有边的权重和

在节点从一个社区移动到另一个社区的过程中，节点的权重和 k_i 不变，而社区的所有点的边的权重和之和在节点移入移出的过程中又很好维护，因此计算复杂度大大降低了。

2.1.2.4 对无向无权图的处理

只有在第一次迭代中才存在无权图（合并节点后一定会引入权重），因此可以把无权图的权重当作 1；无向图则可以表示为双向有向图，但是这样一来所有的边的权重实际上变成了原来的两倍, 因此我将第一次迭代中的自环的权重乘 2 作为修正后的权重（之后的迭代对自环不做乘 2 处理）。

2.2 环境选择及搭建

CPU	AMD EPYC 7K62 48-Core Processor
核心数量	48

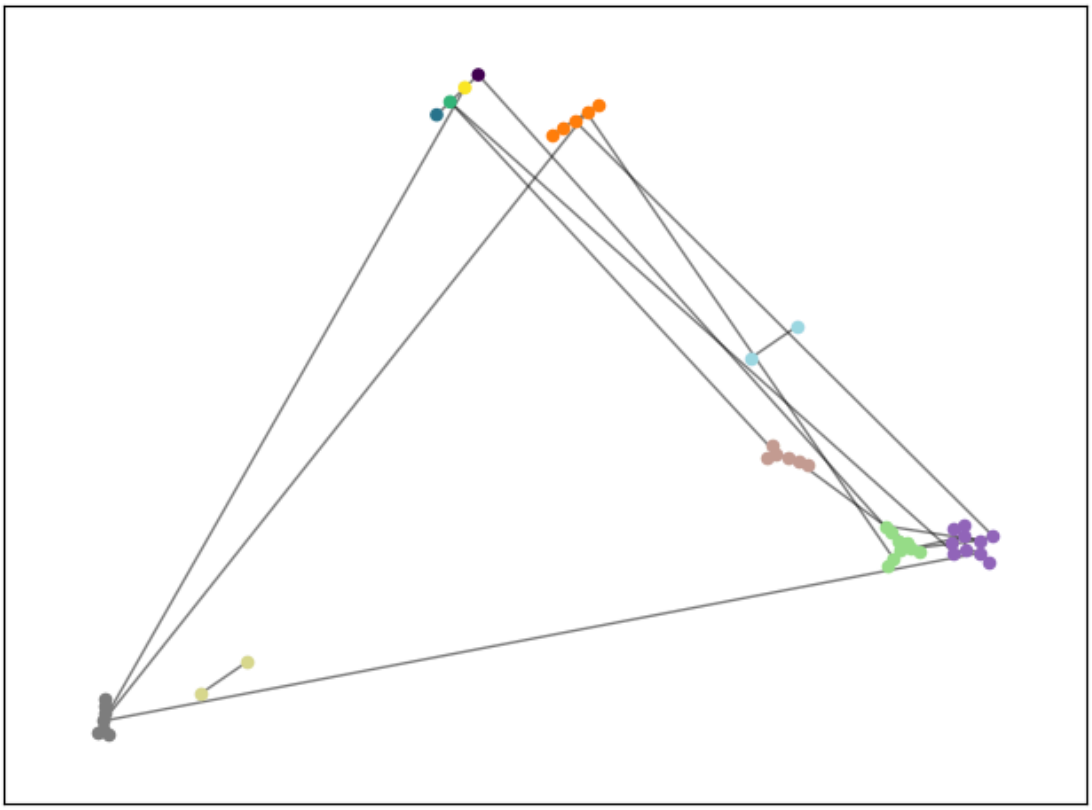
操作系统	Linux
内存	64 GB
内核版本	5.4.241-1-tlinux4-0017.12
rustup 版本	1.27.1

2.2.1 框架与编程语言

为了锻炼自己代码能力，我选择从头开始实现 Louvain 算法，并没有使用图计算框架。而我使用的 Rust 语言则是一门安全、并发、高效的新型语言，拿来写社区发现算法再适合不过了，我也将在后面的深入介绍中证明这一点。

2.2.2 脚本工具

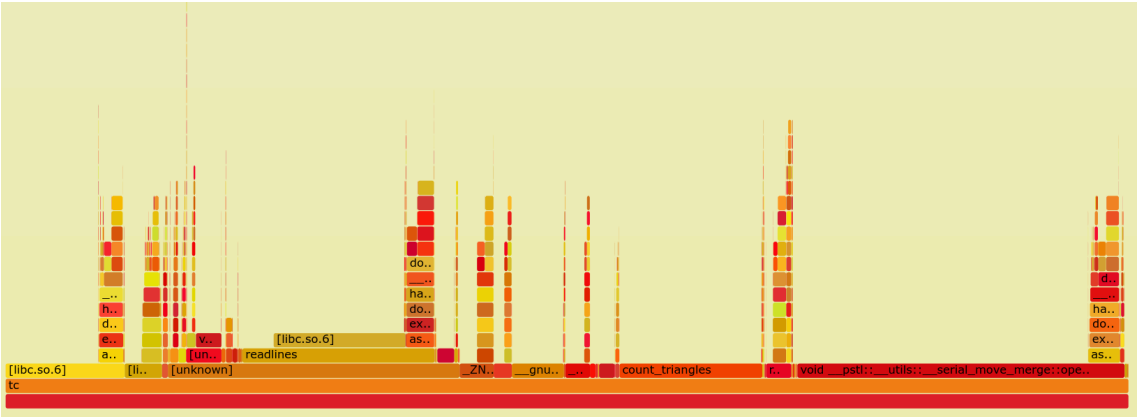
我把编写了两个脚本工具 gen_random_graph.py 和 partition.py。前者用来随机生成较小的随机网络，方便我快速验证算法实现的正确性；后者使用 Python NetworkX 的 Louvain 实现来进行社区划分并进行可视化，它的结果也可以用来验证我自己 Louvain 算法的正确性。



2.2.3 profiling 工具选择

首先使用的是 Linux 经久不衰的 perf 工具和火焰图。

- 每一列代表一个调用栈，每个格子代表一个函数。
- 纵轴展示了栈的深度，从下到上排列，最顶上的格子代表当前占用 CPU 的函数。
- 横轴代表采样中出现频率，宽度越大，说明该函数可能是瓶颈原因的可能性就越大



2.3 优化一：IO 优化

使用 mmap 函数将文件映射到内存中，然后通过字符串切片 content 可以直接访问文件的内容：

```
let file = File::open(filename).unwrap();
let mmap = unsafe { memmap2::Mmap::map(&file).unwrap() };
let content = &mmap[..];
```

将文件映射到内存中可以减少系统调用次数，使用 mmap 后，可以直接通过内存指针对文件进行访问，而不需要每次都调用 read 或 write 系统调用来读写文件。这可以显著减少系统调用的开销，尤其是在这种大文件的场景下。

而且 mmap 将整个文件一次性映射到内存中，而不是像 fstream 那样每次只读取或写入一块数据。这可以更好利用操作系统的文件页缓存机制，使文件的后续访问更快。

2.4 优化二：parse 优化（系统调用与 SIMD 优化）

在读入文件后，不选择使用 scanf 或者 fstream 来解析数据，而是手动使用 memchr 辅以 simd_atoi 等函数来解析。这样提升非常明显，甚至能达到上百倍的加速比。

这是因为 scanf 或者 sscanf 等标准库函数要么是需要大量系统调用，要么是设计出来针对的情况非常多，有大量的时间用来判断各种匹配情况了，实际上只需要简单的处理整数的功能即可。同时，simd_atoi 利用了单指令多数据(SIMD)技术，可以加速整数的解析。

2.5 优化三：并行优化

解析数据和构建邻接表，都没有数据依赖或者竞争的问题，天生适合并行的优化。

2.5.1 并行优化解析数据

首先要明确，当线程数等于核数的时候，性能应该是最好的：既利用了多线程的优势，又节省了频繁调度带来的上下文切换的损失。

可以直接将 content 划分成 CPU 个数个切片，每个线程从距离开始位置最近的新的一行开始，这样可以做到解析时不重不漏。

解析完成后将不同线程的结果合并到一起，这同样可以使用多线程的方法

```
let result = Mutex::new(Vec::new());
let edge_slices = {
    rayon::scope(|s| {
        for i in 0..nthreads {
            let start = splitting_positions[i];
            let end = splitting_positions[i + 1];
            let slice = &content[start..end];
            s.spawn(|_| {
                let edges = read_slice(slice);
                result.lock().unwrap().push(edges);
            });
        }
    });
    result.into_inner().unwrap()
};
```

2.5.2 并行优化构建邻接表

为了避免数据竞争，要先排序，让每个线程处理的节点互不重复，这个排序也是可以并行化的。使用 rust rayon 的进行并行排序，可以将速度近似线性提升。

```
result.voracious_mt_sort(rayon::current_num_threads());
```

并行构建邻接表的过程与并行解析文件的过程类似，就不再赘述。

2.6 优化四：减少重复计算

在计算节点加入相邻社区的增益时，需要遍历节点所有的邻居节点，并找到邻居节点所在的社区计算 modularity gain，这里存在很多重复计算，因为一个节点可能有多个邻居节点属于同一个社区。因此，可以用一个 HashSet 来消除重复计算：

```
fn max_modularity_gain(&self, vertex: &Vertex) -> Option<u32> {
    ...
    let mut calculated = HashSet::new();
    for &neighbor_vertex_id in vertex.neighbors.keys() {
        ...
        if calculated.contains(&neighbor_community_id) {
            continue;
        }
        ...
    }
    ...
}
```

对于顶点数 34546，边数 421578 的图，在减少重复计算后达到了 3x 的加速比。

减少重复计算前	减少重复计算后
7.600 s ± 0.029 s	2.602 s ± 0.198 s

实验结果

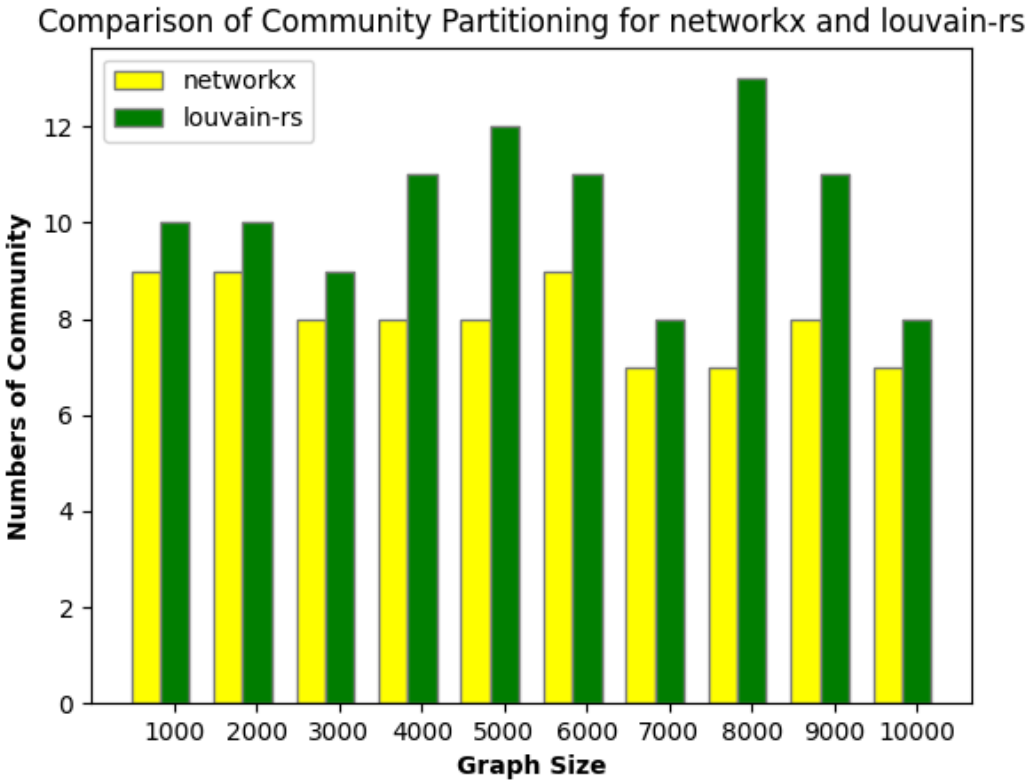
时间上，相对 baseline Python NetworkX 加速比达到了 10x 以上：

	Python NetworkX	louvain-rs
cit-HepPh	25.765 s ± 2.075 s	2.622 s ± 0.040 s
soc-LiveJournal1	运行时间过长	1357.794 s ± 20.653

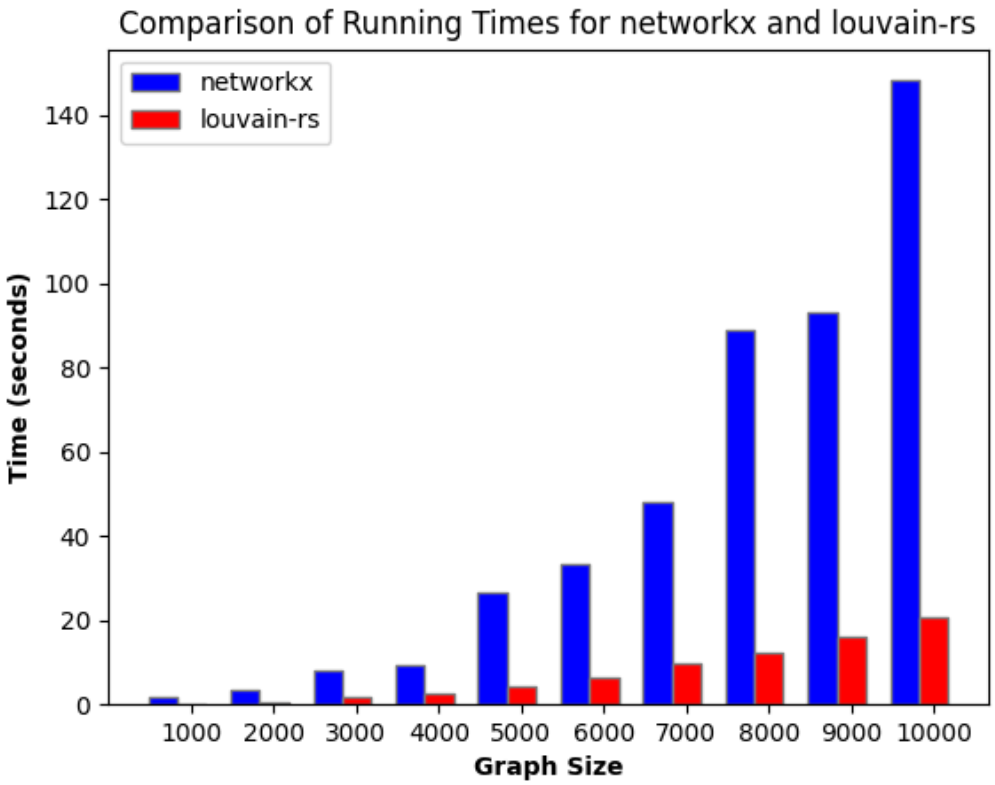
在社区划分效果上, louvain-rs 也强于 Python NetworkX, 对于数据集 cit-HepPh, 在划分社区个数相近的情况下模块度大于 NetworkX，说明其划分效果更好：

	Python NetworkX	louvain-rs
模块度	0.669	0.728
社区个数	83	81

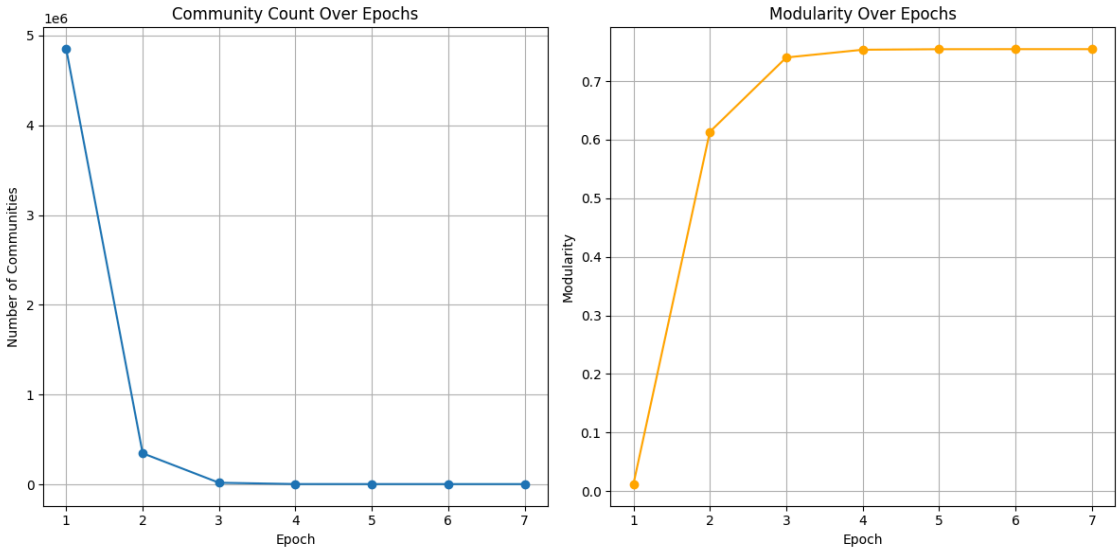
使用脚本随机生成规模在 1000 10000 的网络，分别使用 NetworkX 和 louvain-rs 计算。 louvain-rs 与 NetworkX 计算得到的社区数接近但是更多，说明其划分更细致：



对比 NetworkX 和 louvain-rs 的用时，可以发现 louvain-rs 速度远胜前者：



对于超大图 soc-LiveJournal1 则有如下收敛趋势，可以发现收敛迅速，快速出现了“手肘点”:



实验总结

本次大数据算法综合实践的图计算实验我没有选择使用图计算框架，而是直接使用 Rust 手写，既是想要挑战自我，也是认为在这种较为简单的图计算场景中更贴近底层的实现更有优势。

本次实验锻炼了我的 Rust 和 Linux 系统编程水平，也考验了我的高性能计算的优化技巧，只有综合运用了多种优化方法（IO 优化、减少系统调用、并行优化等）才能有优秀的加速比，才能将求解时间降低到可观的水平。

目前实现的 Louvain 算法的主体部分的并行度不够高，还需要继续学习，提高其并行度，将其改造成一个高拓展的分布式算法！