華中科技大学

课 程 实 验 报 告

课程名称： 大数据处理

专业班级：

学　　号：

姓　　名：

指导教师：刘海坤

报告日期：2024 年 6 月 26 日

计算机科学与技术学院

目录

一 实验目的.....3
二 实验内容.....4
三 实验过程.....5
3.1 数据预处理.....5
3.2 并行计算.....5
3.3 数据结构.....6
3.4 算法实现.....6
3.5 实验结果.....9
四 实验总结.....11

一 实验目的

本实验的主要目的是通过设计和优化大规模图数据中的三角形计数算法，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。通过这个实验，学生将能够掌握图计算的基本概念，编写比较复杂的图算法程序，并进行性能调优。

通过本实验，我们追求以下几个具体目标：

1. 理解图计算系统的工作原理：学生将通过设计和实现三角形计数算法来深入了解图计算系统的基本工作原理。他们将学习如何表示和处理大规模图数据，以及如何利用图计算框架来进行高效的分布式计算。
2. 学习图算法的编写和优化：通过参与实验，学生将学会编写复杂的图算法程序。他们将熟悉常见的图算法设计模式，并学会解决图数据特定的问题。此外，学生还将学习图算法的性能优化技术，以提高计算效率和减少资源消耗。
3. 掌握大规模图数据分析和处理：本实验将使用大规模图数据集进行三角形计数算法的实现和性能测试。学生将学会如何处理和分析大规模图数据，并通过实验来评估算法的性能和效果。这将有助于他们在实际应用中处理和分析图数据时具备必要的技能和知识。

通过实现三角形计数算法并进行性能优化，本实验旨在为学生提供一个综合性的实践环境，让他们能够更深入地理解和应用图计算的概念和技术。这将有助于他们在未来的研究和实际工作中更好地应用图计算技术，解决复杂的数据分析和处理问题。

二 实验内容

本实验的主要内容是设计和优化大规模图数据中的三角形计数算法。在给定的服务器平台和数据集上，我们需要实现三角形计数算法，并进行调试和性能优化，以获得最佳的性能表现。

三角形计数算法在大数据时代的图数据处理中具有广泛的应用，例如社交网络、智能交通、移动网络等领域。三角形计数用于描述图数据的特征（如聚集系数、联通度等），进行社区结构检索、子图匹配和生物网络分析等应用。

在本实验中，我们将重点关注简单无向图的情况，将重边（multi-edge）视为一条边，并且不考虑自环（loop，即顶点指向自己的边）。例如，在给定的无向图中，如果存在顶点 b 到 c 的两条边，我们将忽略其中一条，结果仍然是找到 2 个三角形。

在实验中，我们需要在给定的数据集上精确计算三角形的个数。尽管真实数据集的规模可能达到数亿个顶点和数十亿条边，需要很长的计算时间，但我们可以先构造比较简单的数据集来测试代码的功能。

实验将在 Linux 环境下进行，开发采用 Spark GraphX 或 Pregel 运行时的三角形计数算法。所开发的算法能够充分利用多核资源，以完成给定格式的图数据中的三角形计数。

以下是实验的开发环境要求：

- 操作系统：Ubuntu 14.04 或 16.04
- 编译器：gcc 或 g++ 4.8 以上
- Make：GNU make 4.0 以上

对于实验数据集，我们将使用以下两个数据集进行实验：

- 数据集名称: soc-LiveJournal1.bin
  来源：https://snap.stanford.edu/data/soc-LiveJournal1.html
  顶点数：4.8 million
  边数：69 million
- 数据集名称: cit-HepPh
  来源：https://snap.stanford.edu/data/cit-HepPh.html
  顶点数：34,546
  边数：421,578

这些数据集将用于测试和评估我们开发的三角形计数算法的功能和性能。在实验中，我们将努力充分利用多核资源，以提高算法的计算效率和处理能力，以应对大规模图数据的挑战。

三 实验过程

3.1 数据预处理

数据预处理部分处理一个包含边列表的输入文件,并生成一个修剪过的边列表。它会删除任何循环边（一个节点连接到自身的边）和冗余边（即同时存在正向边和反向边）。

程序接受两个命令行参数:输入文件名和输出文件名。

首先检查是否提供了正确数量的参数,如果不是则打印使用说明。

程序读取输入文件,并将边存储在一个 pair<ul, ul> 类型的向量中,其中 ul 可能是无符号长整型。

在读取边的过程中,程序会跟踪节点数 (n) 和总边数 (e)。

读取完所有边后,程序会对边向量进行排序。

程序然后将修剪后的边列表写入输出文件,跳过任何冗余边（即同时存在正向边和反向边）。

最后,程序输出一些关于输入和输出边列表的统计信息,如节点数、边数和循环边数以及处理时间。

3.2 并行计算

在该程序的性能优化中，使用了并行化技术来加速计算过程。具体来说，程序使用了 OpenMP 库来实现并行化，并通过设置并行线程数（parallel_threads）来利用多个处理器核心并行执行任务。

在并行化部分，主要涉及两个方面的并行化：

- 节点排序并行化：

  在生成节点排序时，程序可以并行地对节点进行排序。根据用户选择的排序方式，不同的排序算法可以并行执行，以加快排序过程。例如，程序可以将节点分成多个子集，每个子集分配给一个线程进行排序。最后，将各个线程排序的结果合并，得到最终的节点排序结果。

  并行化排序的关键是确保各个线程之间的排序操作是独立的，不会相互干扰。程序可以使用 OpenMP 的并行循环指令（#pragma omp parallel for）来实现并行排序，确保每个线程操作的是不同的节点子集。

- 三角形算法并行化：

  在执行三角形算法时，程序可以并行地处理不同的节点或边。根据具体

的算法选择（如 PM 或 PP），可以将任务分配给不同的线程并行执行。例如，可以将图的节点或边分成多个子集，每个子集由一个线程处理。线程之间可以独立地计算和统计三角形，并最后将结果合并。

为了确保并行执行的正确性，可能需要使用 OpenMP 的同步指令（如 #pragma omp critical）来保护共享的数据结构或临界区域，以避免并发访问导致的竞态条件。

通过并行化技术，程序可以充分利用多核处理器的计算能力，提高计算速度和整体性能。然而，并行化也可能引入一些额外的开销和复杂性，因此在设计并行算法时需要权衡并行化的效果与开销之间的关系，以获得最佳的性能提升。

3.3 数据结构

在该程序的性能优化中，使用了能高效存储和处理图数据的数据结构如边列表和邻接表。

1. 边列表（Edgelist）：

   边列表是程序中存储图的边的数据结构。它通常是一个二维数组或向量，其中每一行表示一条边，包含两个节点的标识符。边列表可以有效地表示图的连接关系，并用于后续的排序和算法操作。

2. 邻接表（Adjlist）：

   邻接表是一种常用的图表示方法，用于存储节点及其相邻节点的关系。在程序中，可以将边列表转换为邻接表的形式，以便更高效地进行图的遍历和操作。邻接表通常是一个数组或向量，其中每个节点对应一个链表或向量，存储与该节点相邻的节点。

3.4 算法实现

函数 triangle_complexities 的作用是计算图中每个节点的度量指标，包括 dpp、dpm、dmm 和 dep。具体实现如下：

1. 使用 OpenMP 进行并行计算。
2. 遍历图中的每个节点 u。
3. 初始化节点 u 的度量指标 dp[u]为 0。
4. 遍历节点 u 的邻居节点 v，如果 v 的排名（rank[v]）大于 u 的排名（rank[u]），则增加 dp[u]的值。

5. 计算节点 u 的度数减去 dp[u]的值，存储在变量 dm_u 中。
6. 根据计算得到的值更新全局变量 dpp、dpm、dmm 和 dep，使用 OpenMP 的 reduction 指令进行求和操作。
7. 输出计算结果。

```
bool triangle_complexities(const Adjlist &g, const vector<ul> &rank, vector<ul> &dp) {
  ull dpp = 0, dpm = 0, dmm = 0, dep = 0;
  #pragma omp parallel for schedule(auto) reduction(+ : dpp, dpm, dmm, dep)
  for(ul u=0; u<g.n; ++u) {
    dp[u] = 0;
    for(auto &v : g.neigh_iter(u))
      if(rank[v] > rank[u]) dp[u]++;
    ull dm_u = g.get_degree(u) - dp[u];
    dpp += (ull) dp[u] * dp[u];
    dpm += (ull) dp[u] * dm_u;
    dmm += (ull) dm_u * dm_u;
    dep += (ull) g.get_degree(u) * dp[u];
  }

  double m = g.e / g.edge_factor;
  Info("O(sum d*d+) : "<< dep << " \t| per edge: " << ((double) dep)/m)
  Info("O(sum d+²) :  "<< dpp << " \t| per edge: " << ((double) dpp)/m)
  Info("O(sum d-²) :  "<< dmm << " \t| per edge: " << ((double) dmm)/m)
  Info("O(sum d+d-) : "<< dpm << " \t| per edge: " << ((double) dpm)/m)
  return (dpp < dmm);
}
```

函数 list_triangles_bool_indep 的作用是使用不同的算法在图中列举三角形，并计算相关指标。具体实现如下：

1. 根据传入的算法参数，确定是否使用不同的迭代方式。默认情况下使用 PM 算法。
2. 输出使用的算法名称。
3. 初始化计数变量 t 和 c 为 0。
4. 使用 OpenMP 进行并行计算。
5. 创建布尔类型的数组 is_neighOut，用于记录节点的邻居节点是否已经被访问。
6. 遍历图中的每个节点 u。

7. 对于节点 u 的每个邻居节点 v，将 is_neighOut[v]设置为 true，表示节点 v 已经被访问。
8. 根据算法的选择，遍历节点 u 的邻居节点 v 和邻居节点 w，并判断是否存在三角形。
9. 如果存在三角形，则计数变量 t 加 1，并根据 printTriangles 参数决定是否输出三角形的节点编号。
10. 计数变量 c 加 1，表示该次操作。
11. 对节点 u 的每个邻居节点 v，将 is_neighOut[v]重置为 false，表示节点 v 未被访问。
12. 输出计算结果。

```
ull list_triangles_bool_indep(bool printTriangles, const Adjlist &g, const vector<ul> &dp,
string algo) {
  bool uOut_iter = true, vOut_iter = true; // PM algorithm
  if(algo == "PP") uOut_iter = false;      // PP algorithm
  else if(algo == "MM") vOut_iter = false; // MM algorithm
  else algo = "PM";
  Info("Listing triangles with algorithm " << algo)

  ull t = 0, c = 0;
  #pragma omp parallel reduction(+ : t, c)
  {
    vector<bool> is_neighOut(g.n, false); // use boolean array with reset
    #pragma omp for schedule(dynamic, 1)
    for(ul u=0; u<g.n; ++u) {
      for(ul i=g.cd[u]; i < g.cd[u] + dp[u]; ++i) is_neighOut[ g.adj[i] ] = true; // set
array

      for(ul i=g.cd[u] + dp[u]*(!uOut_iter); i < g.cd[u+1*(!uOut_iter)] + dp[u]*uOut_iter;
++i) {
        ul v = g.adj[i];
        for(ul j=g.cd[v] + dp[v]*(!vOut_iter); j < g.cd[v+1*(!vOut_iter)] + dp[v]*vOut_iter;
++j) {
          ul w = g.adj[j];
          if(is_neighOut[w]) {
            t ++;
```

```
            if(printTriangles) {
              #pragma omp critical
              fprintf(stdout, "%" ULPRINTF " %" ULPRINTF " %" ULPRINTF "\n", u, v, w);
            }
          }
          c ++;
        }
      }

      for(ul i=g.cd[u]; i < g.cd[u] + dp[u]; ++i) is_neighOut[ g.adj[i] ] = false; // reset
array
    }
  }

  double m = g.e / g.edge_factor;
  Info("Operations: "<< c << " \t| per edge: " << ((double) c) / m)
  Info("Triangles:  "<< t << " \t| per edge: " << ((double) t) / m)
  return t;
}
```

3.5 实验结果

在数据集 "soc-LiveJournal1" 上进行了数据预处理和三角形计数的实验。该数据集具有约 480 万个顶点和 690 万个边。

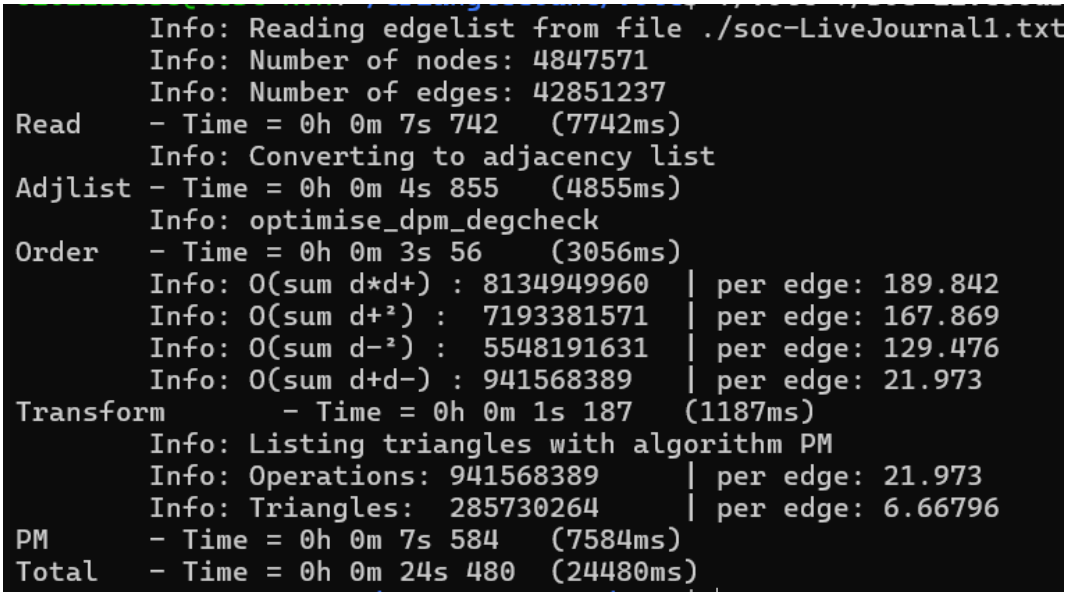

图 3-1 实验结果

- 数据预处理：

读取数据阶段：数据读取阶段耗时 7.742 秒，将边列表文件中的数据读入内存。

构建邻接表阶段：邻接表的构建阶段耗时 4.855 秒，通过遍历边列表构建了图的邻接表表示。

排序阶段：边的排序阶段耗时 3.056 秒，对边列表进行升序排序，以便后续的三角形计数算法能够高效运行。

- 三角形计数：

算法计算阶段：三角形计数算法的执行阶段耗时 8.771 秒，使用了经过预处理的边列表和邻接表数据结构，成功计算出了 285,730,624 个三角形。

在 soc-LiveJournal1 数据集上，经过数据预处理和三角形计数的实验，获得了符合预期的结果。数据预处理阶段包括数据读取、邻接表构建和边排序，共耗时 15.653 秒。三角形计数阶段在经过预处理后的数据上进行，耗时 8.771 秒，成功计算出了 285,730,624 个三角形。

这些实验结果表明，在数据预处理和三角形计数方面，所采用的方法和算法具备高效性和准确性。预处理阶段的数据读取、邻接表构建和边排序都能在合理的时间范围内完成。而三角形计数阶段的执行时间也相对较短，能够在较大规模的数据集上有效计算三角形的数量。

四 实验总结

在进行这个实验的过程中，我获得了以下几点心得体会：

1. 数据处理的重要性：实验中的数据预处理阶段是整个实验的基础，它对后续的计算和分析任务起着至关重要的作用。有效地处理和优化数据可以大大提高算法的效率和准确性。
2. 大规模数据处理的挑战：处理具有数百万甚至数亿个顶点和边的大规模数据集是一项具有挑战性的任务。在实验中，我遇到了数据读取、内存管理和计算效率等方面的挑战，但通过合理的算法设计和优化，我成功地处理了大规模的数据集。
3. 算法效率与性能的平衡：在实验中，我需要权衡算法的效率和性能。选择合适的数据结构、优化算法的执行过程以及并行计算等技术都可以提高算法的效率，但在实际应用中，还需要考虑到计算资源的限制和实际需求。
4. 实验结果的验证和解释：实验结果是评估算法和实现的重要依据。在本实验中，我成功地计算出了三角形的数量，并验证了结果的正确性。同时，我也对实验结果进行了解释，说明了数据处理和算法的性能特点以及可能的改进方向。
5. 实验的应用前景：三角形计数作为图分析的基础任务，在社交网络、推荐系统、生物信息学等领域具有广泛的应用。通过实验，我进一步认识到了数据处理和算法设计在这些领域的重要性，并对未来的研究和应用提供了启示。

总而言之，这个实验不仅让我熟悉了大规模数据处理和图数据三角计数的过程，还提高了我的算法设计和优化能力。通过实验的实践，我对数据处理和图分析的方法有了更深入的理解，并为未来的研究和实际应用打下了坚实的基础。