《时空大数据高性能处理》上机实验

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 章节 | **第三次上机：仿射变换拟合** | | | | | 序号 | 3 |
| 姓 名 | 甘劲博 | 系院专业 | 计算机 | 班级 | 21计科5 | 学 号 | 2110551202 |
| 日期 | 2024-10-10 | | 指导教师 | 刘舟 | | 成 绩 |  |
| **一、内容**  在理解掌握最小二乘法进行线性回归的基础上，学习二维图形的仿射变换模型。思考给定若干个二维平面点对后（初始坐标-仿射变换坐标），如何编程实现仿射变换模型的拟合。分析已知点数量多少对上述程序执行性能的影响。 | | | | | | | |
| **2.1主要c/c++代码**  包含：仿射变换拟合  开源地址：<https://github.com/Duxingmengshou/HPCLab>  仿射变换拟合：  #include <iostream>  #include <vector>  #include <omp.h>    struct Point {  double x;  double y;  };    void fitAffineTransform(const std::vector<Point>& original, const std::vector<Point>& transformed, double& a, double& b, double& c, double& d, double& tx, double& ty) {  int n = original.size();    // 确保原始点和变换点数量一致  if (n != transformed.size() || n < 3) {  std::cerr << "Point sets must be of the same size and at least 3 points are required." << std::endl;  return;  }    double sum\_x = 0, sum\_y = 0, sum\_x\_prime = 0, sum\_y\_prime = 0;  double sum\_xy = 0, sum\_x2 = 0, sum\_y2 = 0;    // 并行计算各个求和项  #pragma omp parallel for reduction(+:sum\_x, sum\_y, sum\_x\_prime, sum\_y\_prime, sum\_xy, sum\_x2, sum\_y2)  for (int i = 0; i < n; ++i) {  double x = original[i].x;  double y = original[i].y;  double x\_prime = transformed[i].x;  double y\_prime = transformed[i].y;    sum\_x += x;  sum\_y += y;  sum\_x\_prime += x\_prime;  sum\_y\_prime += y\_prime;  sum\_xy += x \* y\_prime;  sum\_x2 += x \* x;  sum\_y2 += y \* y;  }    // 计算均值  double mean\_x = sum\_x / n;  double mean\_y = sum\_y / n;  double mean\_x\_prime = sum\_x\_prime / n;  double mean\_y\_prime = sum\_y\_prime / n;    // 计算变换参数  a = (sum\_xy - n \* mean\_x \* mean\_y\_prime) / (sum\_x2 - n \* mean\_x \* mean\_x);  b = (sum\_y\_prime - mean\_y\_prime - a \* mean\_x) / mean\_y;  c = (sum\_y\_prime - mean\_y\_prime - a \* mean\_x) / mean\_x;  d = (sum\_y - mean\_y) / mean\_y;    // 计算平移量  tx = mean\_x\_prime - a \* mean\_x - b \* mean\_y;  ty = mean\_y\_prime - c \* mean\_x - d \* mean\_y;  }    int main() {  std::vector<Point> original = {{0, 0}, {1, 1}, {2, 2}};  std::vector<Point> transformed = {{1, 2}, {2, 3}, {3, 6}};    double a, b, c, d, tx, ty;  fitAffineTransform(original, transformed, a, b, c, d, tx, ty);    std::cout << "Affine Transform Parameters:" << std::endl;  std::cout << "a: " << a << ", b: " << b << ", c: " << c << ", d: " << d << ", tx: " << tx << ", ty: " << ty << std::endl;    return 0;  }  CMakeList.txt：  cmake\_minimum\_required(VERSION 3.26)  project(openmp\_check)    set(CMAKE\_CXX\_STANDARD 17)    add\_executable(${PROJECT\_NAME} main.cpp)    target\_compile\_options(${PROJECT\_NAME}  PRIVATE  /openmp  )    target\_link\_libraries(${PROJECT\_NAME} ${pthreads\_ROOT\_LIBRARIES}) | | | | | | | |
| **2.2运行截图**  仿射变换拟合：  已知点数量多少对上述程序执行性能的影响：  程序的计算复杂度主要是线性时间复杂度(O(n))，这意味着随着已知点数量的增加，计算所需的时间也会线性增加。当处理的数据点较多时，执行时间会显著增长，这在大规模数据处理时尤为明显。  内存使用也是一个不可忽视的因素。随着点数量的增加，程序需要存储更多的Point结构体实例，导致内存使用量线性增长。此外，如果数据点存储在外部（例如文件或数据库），在处理大量数据时，数据的传输开销可能会显著影响整体性能。  并行化效率同样是影响程序性能的重要因素。在当前实现中，使用OpenMP来并行化求和操作。对于较小的数据集，线程创建和管理的开销可能会超过并行计算带来的速度提升，但随着点数量的增加，线程的利用率提高，性能提升效果开始显现。此外，在数据量较大时，OpenMP能够有效分配任务，提高整体计算效率，这对于处理大规模数据至关重要。  数据局部性也需要考虑。对于较小的数据集，数据可能完全存放在 CPU 缓存中，导致快速访问。但随着数据集增大，可能会出现缓存未命中，进而影响性能。同时，当数据量增大时，内存带宽可能成为瓶颈，限制了程序的执行速度。  在实际应用中，性能的提升需要通过测试来验证。基准测试是一个有效的方法，可以使用不同数量的点（例如 100、1000、10000、100000 等）进行测试，测量每次运行的时间，并观察随着点数量的增加，程序执行时间的变化趋势。这有助于评估并行化的有效性，找出最佳的性能平衡点。  对于小规模数据，程序的执行时间主要由计算复杂度决定，并可能受到并行化开销的影响。随着点数量的增加，程序性能逐渐提升，OpenMP的并行化效果开始显现。当点数量达到数千或数万时，程序性能会显著提高，内存使用和并行计算的优势开始明显。因此，合理选择点数量和优化并行化策略，可以显著提升程序性能。 | | | | | | | |