《时空大数据高性能处理》上机实验

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 章节 | **第二次上机：最小二乘法算法** | | | | | 序号 | 2 |
| 姓 名 | 甘劲博 | 系院专业 | 计算机 | 班级 | 21计科5 | 学 号 | 2110551202 |
| 日期 | 2024-10-03 | | 指导教师 | 刘舟 | | 成 绩 |  |
| **一、内容**  理解最小二乘法的原理，掌握如何用最小二乘法进行线性回归。并思考给定若干个二维平面点时，如何编程实现上述线性回归。（即根据若干个点x，y值，拟合直线：y=kx+b）在此基础上，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：最小二乘法的pthread实现  开源地址：<https://github.com/Duxingmengshou/HPCLab>  最小二乘法的pthread实现：  #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>    typedef struct  {  double \*x;  double \*y;  int start;  int end;  double \*S\_x;  double \*S\_y;  double \*S\_xy;  double \*S\_xx;  } ThreadData;    void \*calculate\_sums(void \*arg)  {  ThreadData \*data = (ThreadData \*)arg;  double local\_S\_x = 0.0, local\_S\_y = 0.0, local\_S\_xy = 0.0, local\_S\_xx = 0.0;    for (int i = data->start; i < data->end; i++)  {  local\_S\_x += data->x[i];  local\_S\_y += data->y[i];  local\_S\_xy += data->x[i] \* data->y[i];  local\_S\_xx += data->x[i] \* data->x[i];  }    // 合并结果  \*data->S\_x += local\_S\_x;  \*data->S\_y += local\_S\_y;  \*data->S\_xy += local\_S\_xy;  \*data->S\_xx += local\_S\_xx;    return NULL;  }    void linear\_fit(double \*x, double \*y, int n, int num\_threads, double \*k, double \*b)  {  pthread\_t threads[num\_threads];  ThreadData thread\_data[num\_threads];  double S\_x = 0.0, S\_y = 0.0, S\_xy = 0.0, S\_xx = 0.0;    int chunk\_size = n / num\_threads;    // 创建线程  for (int i = 0; i < num\_threads; i++)  {  thread\_data[i].x = x;  thread\_data[i].y = y;  thread\_data[i].start = i \* chunk\_size;  thread\_data[i].end = (i == num\_threads - 1) ? n : (i + 1) \* chunk\_size;  thread\_data[i].S\_x = &S\_x;  thread\_data[i].S\_y = &S\_y;  thread\_data[i].S\_xy = &S\_xy;  thread\_data[i].S\_xx = &S\_xx;    pthread\_create(&threads[i], NULL, calculate\_sums, &thread\_data[i]);  }    // 等待所有线程完成  for (int i = 0; i < num\_threads; i++)  {  pthread\_join(threads[i], NULL);  }    // 计算斜率 k 和截距 b  \*k = (n \* S\_xy - S\_x \* S\_y) / (n \* S\_xx - S\_x \* S\_x);  \*b = (S\_y - (\*k) \* S\_x) / n;  }    int main()  {  int n = 10; // 数据点数量  double x[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  double y[] = {2.2, 2.8, 3.6, 4.5, 5.1, 6.3, 7.8, 8.5, 9.1, 10.2};    double k, b;  int num\_threads = 4; // 线程数量    linear\_fit(x, y, n, num\_threads, &k, &b);    printf("拟合直线: y = %.2fx + %.2f\n", k, b);  return 0;  }  Makefile：  MPICXX = gcc  CXXFLAGS = -Wall -O2    all: 1-s    1-s: 1-s.c  $(MPICXX) $(CXXFLAGS) -o 1-s 1-s.c    clean:  rm -f 1-s | | | | | | | |
| **2.2运行截图**    最小二乘法的pthread实现：    分析已知点数量多少对上述程序执行性能的影响：  在使用pthread进行线性回归的多线程实现时，随着数据点数量的增加，计算所需的和的复杂度为O(n)，这意味着处理的数据量越大，计算时间线性增加。在多线程实现中，通常会将数据分成多个块，每个线程处理一个块。理想情况下，线程数量应与数据点数量成正比，以充分利用并行计算的优势。  线程管理和上下文切换也是影响性能的重要因素。每个线程的创建和销毁都有一定的开销，对于小规模数据集，线程管理的开销可能会超过并行计算带来的性能提升。因此，适合将数据点数量与线程数量相匹配，以避免频繁的线程创建和销毁。当线程数量超过CPU核心数时，系统会进行上下文切换，这会增加额外的开销。对于较大的数据集，适当的线程数量（通常与 CPU 核心数相等或略多）可以减少上下文切换，提高性能。  负载均衡也是一个关键因素。如果数据点的数量不均匀分布，可能导致某些线程处理的数据量过大，而其他线程则相对较少，这种不均衡会导致某些线程空闲等待，从而降低整体性能。合理的负载均衡策略（如动态分配任务）可以显著提高效率。在划分数据时，选择合适的块大小也很重要，过小的块可能导致线程管理开销过大，而过大的块可能导致负载不均衡。  随着数据点数量的增加，浮点数计算可能引入数值误差，尤其是在计算斜率k和截距b时。多线程计算时，确保合并结果的顺序和方法可以影响最终结果的精度，因此在设计多线程算法时，需考虑数值稳定性，以减少误差的累积。  性能测试与评估在实际应用中至关重要。监测不同数据规模下程序的执行时间和资源使用情况，可以通过实验评估在不同数据点数量和线程数量下的性能表现，找到最佳的配置。通常，对于较小的数据集，单线程实现可能更快，而当数据点数量达到一定阈值后，多线程实现的优势将逐渐显现。通过实验，可以确定何时应该切换到多线程实现，以获得最佳性能。 | | | | | | | |