自动化专业大数据与算法课程设计报告

Project Report for Big Data and Algorithms

多机-多线程-SSE运算加速

Multi-PC-multi-threads-SSE Speed Up

|  |  |  |  |
| --- | --- | --- | --- |
| 学 院 | 电子与信息工程学院 | | |
| 专 业 | 自动化 | | |
| 学生姓名 | 郭展羽 | 学 号 | 1851170 |
| 学生姓名 | 刘沛江 | 学 号 | 1853318 |
| 指导教师 |  | 王晓年 |  |

2021年 1月8日

目 录

[1. 加速原理 3](#_Toc61032355)

[1.1. 多线程 3](#_Toc61032356)

[1.2. 多机通信与并行 5](#_Toc61032357)

[1.3. SSE指令集 7](#_Toc61032358)

[2. 架构设计 9](#_Toc61032359)

[3. 代码实现 11](#_Toc61032360)

[3.1. 公用部分 11](#_Toc61032361)

[3.2. 单机单线程 18](#_Toc61032362)

[3.3. 单机多线程 19](#_Toc61032363)

[3.4. 多机多线程 21](#_Toc61032364)

[4. 测试与结果分析 25](#_Toc61032365)

[4.1. 测试环境 25](#_Toc61032366)

[4.2. 测试结果 25](#_Toc61032367)

[4.3. 结果分析 31](#_Toc61032368)

[4.4. 附加测试与结果分析 31](#_Toc61032369)

[5. 总结 38](#_Toc61032370)

[5.1. 课程设计总结 38](#_Toc61032371)

[5.2. 课程总结 39](#_Toc61032372)

[6. 重现时注意 40](#_Toc61032373)

[6.1. 编译 40](#_Toc61032374)

[6.2. IP地址更改 40](#_Toc61032375)

[6.3. 运算数据量更改 41](#_Toc61032376)

[6.4. 排序功能数据量更改 41](#_Toc61032377)

[6.5. socket传输数据量更改 42](#_Toc61032378)

[7. 分工 44](#_Toc61032379)

[8. 参考文献 44](#_Toc61032380)

# 加速原理

本次大作业使用到的加速方法有：多线程、多机和SSE指令集，下文将依次对其进行简要的原理介绍与分析。

## 多线程

### 多线程及相关概念简介

#### 进程

进程指的是正在运行的程序。确切的来说，当一个程序进入内存运行，即变成一个进程，进程是处于运行过程中的程序，并且具有一定独立功能。当在撰写这部分内容时，我的计算机上的进程如下图所示：

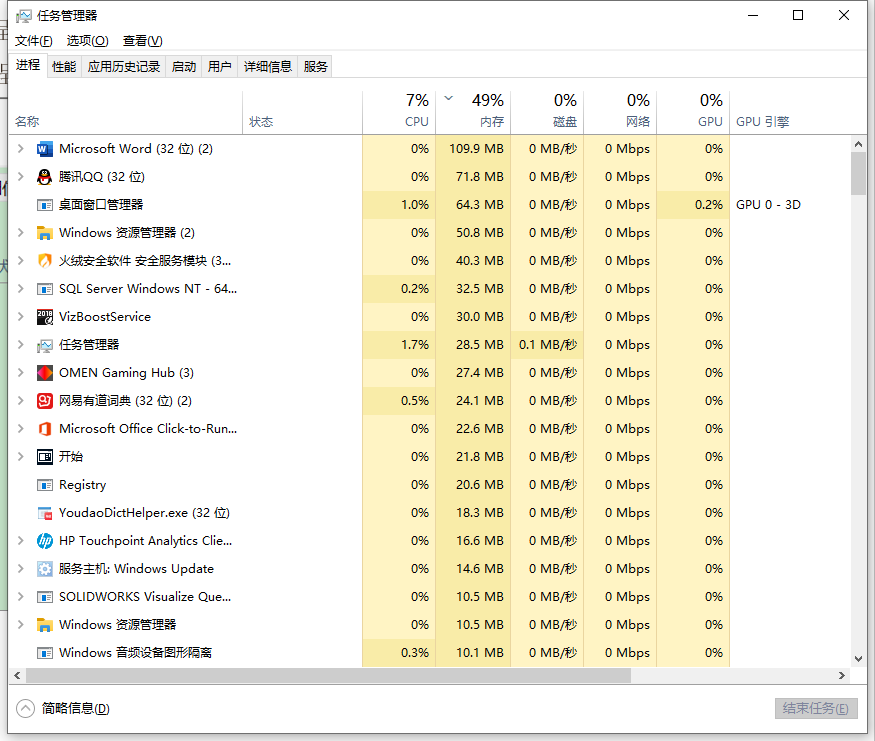


图1-1 计算机上的进程

#### 线程

线程指的是进程中的一个执行单元，负责当前进程中程序的执行。一个进程中至少有一个线程，也可以有多个线程。这样的应用程序也可以称为多线程程序。比如此时我同时用TIM聊天和查看图片，在任务管理器可以看到一个进程有两个线程同时运行，如下图所示：

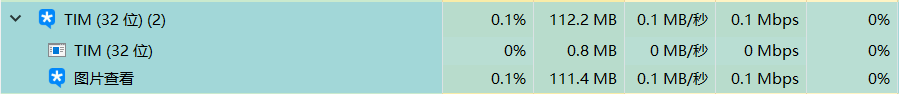


图1-2 一个进程的多个线程

#### 多线程

多线程是指程序中包含多个执行流，即在一个程序中可以同时运行多个不同的线程来执行不同的任务。如图所示，主线程可以和许多工作线程并发执行。

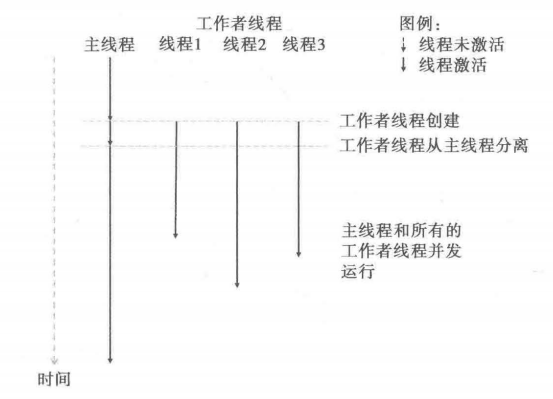


图1-3 多线程例子

### 多线程加速原理

#### 分时调度

所有线程轮流使用 CPU 的使用权，平均分配每个线程占用 CPU 的时间。

#### 抢占式调度

优先让优先级高的线程使用 CPU，如果线程的优先级相同，那么会随机选择一个线程。

CPU（中央处理器）使用抢占式调度模式在多个线程间进行着高速的切换。对于CPU的一个核而言，某个时刻，只能执行一个线程，而 CPU的在多个线程间切换速度相对我们的感觉要快，看上去就是在同一时刻运行。

其实，多线程程序并不能提高程序的运行速度，但能够提高程序运行效率，让CPU的使用率更高。

#### 主线程

主线程执行算法的顺序部分，当遇到需要进行并行计算式，主线程派生出（创建或者唤醒）一些附加线 程。在并行区域内，主线程和这些派生线程协同工作，在并行代码结束时，派生的线程退出或者挂起，同时控制流回到单独的主线程中，称为汇合。

如图所示：

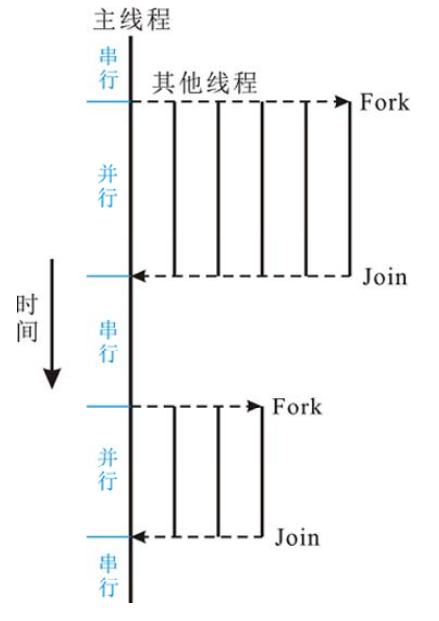


图1-4 主线程与其他线程的关系

所以在多线程程序中，当主线程必须等待的时候，CPU可以运行其它的工作线程，而不是等待，这样就大大提高了程序的效率 。这也是多线程的加速原理。

## 多机通信与并行

### 多机并行计算框架

在物理层面上，多台计算机组成的集群使用低延迟的网络连接起不同的计算节点。针对于本次课程设计所给出的科学计算，先经过对任务进行划分，接着分割到不同的计算集群去执行，最后再完成任务汇总的几个阶段。

### 多机通信

#### Socket介绍

Socket是应用层与TCP/IP协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。

#### UDP

UDP 是不具有可靠性的数据报协议。细微的处理它会交给上层的应用去完成。在 UDP 的情况下，虽然可以确保发送消息的大小，却不能保证消息一定会到达。因此，应用有时会根据自己的需要进行重发处理。

#### TCP

TCP 是面向连接的、可靠的流协议。流就是指不间断的数据结构，当应用程序采用 TCP 发送消息时，虽然可以保证发送的顺序，但还是犹如没有任何间隔的数据流发送给接收端。TCP 为提供可靠性传输，实行“顺序控制”或“重发控制”机制。此外还具备“流控制（流量控制）”、“拥塞控制”、提高网络利用率等众多功能。

借助于socket的网络通信如图所示：

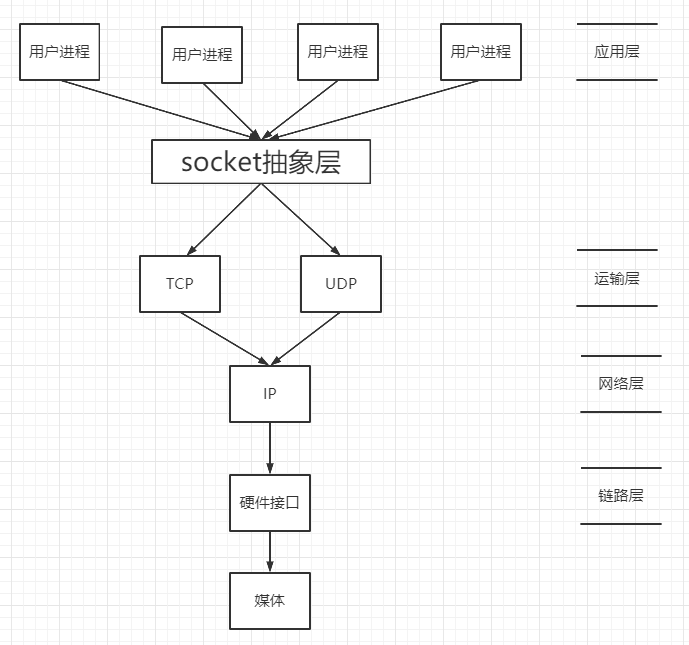


图1-5 socket网络通信

### 多机并行计算加速原理

在本次课程设计中，我们使用两台电脑分别充当服务端和客户端，先由服务端平均分配任务并将数据通过socket抽象层和TCP运输层传送给客户端，当客户端得到计算结果之后再通过socket抽象层和TCP运输层传送给服务端。这是真正的并行计算，使用两份计算资源去执行计算任务，所以如果忽略传输时间，那么计算时间将会减少为单机计算时间的一半。具体流程图如图所示：

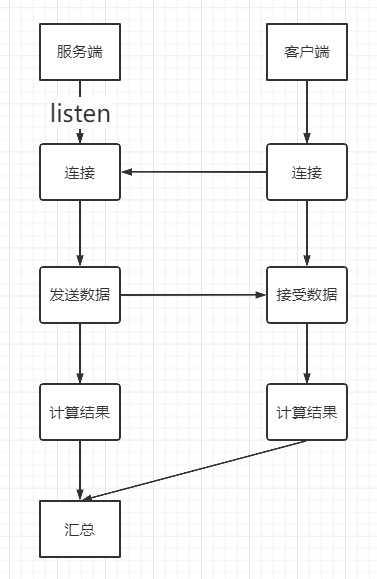


图1-6 应用的流程图

## SSE指令集

### SSE指令集简介

SSE指令集全称为Streaming SIMD Extensions，即流式单指令多数据扩展。这种体系结构对不同的多组数据采用相同指令来处理，比较适合并行算法的实现。其体系结构如下图所示：

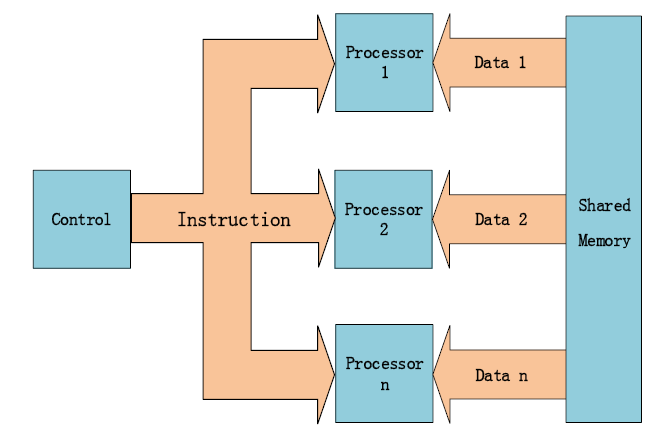


图1-7 SSE体系结构

指令集使用单独的128bit寄存器（XMM寄存器），寄存器个数16（不同计算机可能不同），一次处理128bit的数据。寄存器结构如下：

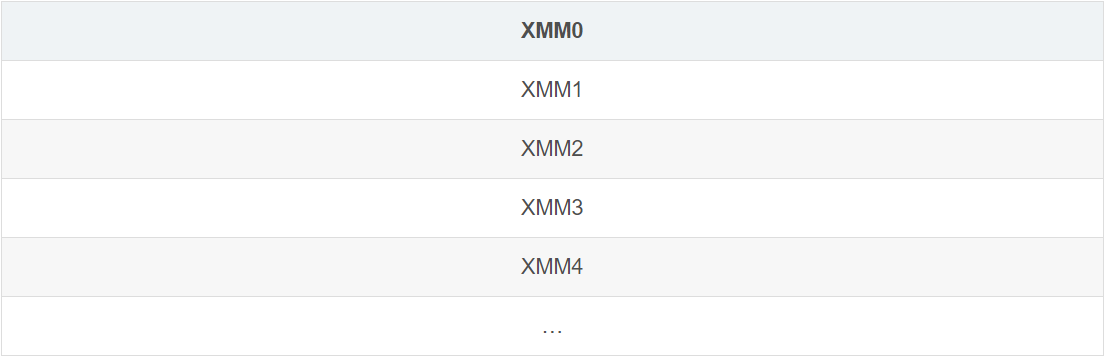


图1-8 寄存器结构

因为一个寄存器的大小为128bit，那么使用SSE指令集一次就可以选择处理2个64bit的数据类型：



或者一次处理4个32bit的数据类型：



### SSE指令集加速原理

正是因为SSE指令集可以一次性处理4个32bit的数据类型，我们就可以使用它来一次性处理4个float类型的数据，我们以\_mm\_add\_ps()函数为例来演示SSE的加速原理，如图所示：



图1-9 运算对比

1. 一条普通加法指令调用一组数据(X, Y)进行加法操作，得到结果XopY。
2. 一条SSE、AVX指令调用四组数据(X1, Y1)、(X2, Y2)、(X3, Y3)、(X4, Y4)，得到结果X0opY0、X1opY1、X2opY2、X3opY3。

所以使用SSE指令处理float型数据，一条指令就可以得到四倍于普通指令的结果。如果CPU执行一条指令的时间相同，那么此时SSE指令的效率就可以达到普通指令的四倍。这也是为什么使用SSE指令集可以实现算法加速的原理。

# 架构设计

本次大作业的单机的软件架构设计如下图所示。



图2-1 单机软件架构设计

可以从图中看出，单机不论是客户端还是服务端，都依次进行了单机单线程、单机多线程、多机多线程的求和、求最大值、排序的计算，并都显示了结果，用以对比耗时长短，具体测试见“4. 测试与结果分析”。而多机通过网络进行协作计算的流程架构如下图所示。



图2-2 多机流程架构

可以从图中看到，不论是服务端运算比较快，还是客户端运算比较快，都能正确接收对方传来的数据。要注意在多机通信的时候，TCP传输的一个包最大为1500字节，因此本次大作业在传输最终排序结果的时候，将客户端的结果分割成1000字节的块，依次发送。将在后续测试中阐述我们遇到的问题及解决办法。

# 代码实现

本次大作业使用Linux系统进行编程，具体的版本为Ubuntu 16.04 LTS，因此代码的形式与Windows存在着差异。

代码实现可以分为三个部分：单机单线程、单机多线程和多机多线程。而在运算开始前，有许多宏定义，和许多共用的函数，将首先进行说明。

## 公用部分

为方便调试时更改，客户端和服务端共同使用了一个头文件“mySrvClt.h”，里面声明定义了两者均会使用的变量、函数。

### 网络相关部分

1. // 网络相关
2. #define PORT        8888            // 端口号：8888
3. #define BUF\_LEN     1024            // 文字缓存长度：1kB
5. #ifdef  CLIENT
6. #define SERVER\_IP   "127.0.0.1"     // 服务端IP地址
7. #endif
9. // 网络收发缓存
10. **char** buf[BUF\_LEN];

定义了网络端口号为8888，网络收发文本消息的缓存大小设置为1024字节。如果为客户端，则还需定义服务器的IP地址，用以绑定进行连接。

### 多线程相关部分

1. // 线程相关
2. #define MAX\_THREADS     64          // 线程数：64
3. #define SUBDATANUM      2000000     // 子块数据量：2000000
4. #define SRV\_SUBDATANUM  1000000     // 单PC数据
5. #define CLT\_SUBDATANUM  1000000     // 单PC数据
6. #define DATANUM         (SUBDATANUM\*MAX\_THREADS)  // 总数据量：线程数x子块数据量
8. // 计算变量
9. **double** rawDoubleData[DATANUM];      // 原始数据
10. **double** doubleResults[MAX\_THREADS];  // 各线程结果
11. **double** finalSum;                    // 求和最终结果
12. **double** finalMax;                    // 求最大值最终结果

定义了线程数为64，每个子块的数据量为2,000,000。在客户端和服务端协作计算的时候，要将该值减半。总的数据量 = 线程数×子块数据量。用以上的宏定义，定义全局变量，分别用于存储原始数据、各线程结果（求和、最大值）、求和最终结果和求最大值最终结果。之所以未提及到排序，是因为需要将排序的各种参数与前面两者分离开来，便于数据的管理和代码的调试（由于排序的时间较长，为了方便调试可以在不影响前两者的情况下减小数据量）。具体的定义如下所示。

1. #define S\_SUBDATANUM        200         // 减少数据量进行排序
2. #define S\_SRV\_SUBDATANUM    100         // 单PC小数据量测试
3. #define S\_CLT\_SUBDATANUM    100         // 单PC小数据量测试
4. #define S\_DATANUM           (S\_SUBDATANUM\*MAX\_THREADS)      // 总数据量(小)
5. #define S\_CLT\_DATANUM       (S\_CLT\_SUBDATANUM\*MAX\_THREADS)  // CLT数据量
7. #ifdef SERVER
8. #define S\_SRV\_DATANUM       (S\_SRV\_SUBDATANUM\*MAX\_THREADS)  // SRV数据量
9. #endif
11. #define S\_ONCE              100                         // 一次发送100个
12. #define S\_TIMES             (S\_CLT\_DATANUM/S\_ONCE)      // 发送100个的次数
13. #define S\_LEFT              (S\_CLT\_DATANUM%S\_ONCE)      // 发送剩余不足数据
15. #define DATA\_MAX            2147483647                  // 随机数的最大值
17. **double** S\_rawDoubleData[S\_DATANUM];              // 排序原始数据
18. **double** S\_sortDoubleData[S\_DATANUM];             // 排序最终结果
19. **double** S\_CLT\_sortDoubleData[S\_CLT\_DATANUM];     // CLT排序最终结果
20. **double** S\_threadResult[MAX\_THREADS][S\_SRV\_SUBDATANUM];// 单机多线程各线程结果
21. **double** S\_threadResult0[MAX\_THREADS][S\_SUBDATANUM];   // 多机多线程各线程结果
23. #ifdef SERVER
24. **double** S\_SRV\_sortDoubleData[S\_SRV\_DATANUM];     // SRV排序最终结果
25. #endif

使用小数据进行测试，类似于大数据的定义。同时因为传输排序结果的时候，需要分块传输，因此将数据每次传输的长度、传输的次数和最后一次剩余数据的传输相关数量先计算完成。因为数据的生成是以随机数的形式，因此将最大值记下，为2147483647，这也是rand()返回的int这种数据类型的最大值（231 - 1）。（Linux中，C语言随机数生成范围不同于在Windows系统，范围为[-INT\_MAX+1, INT\_MAX]，所以在生成数据的时候要取绝对值。）

同样地，定义了各种全局变量：排序原始数据、最终结果、客户端排序的结果、服务端排序的结果、多线程中各线程排序的结果。

1. // 标志位
2. **bool** thread\_begin;          // 线程发令标志

为了保证多线程运算时间的统计，而Linux中的多线程没有像Windows中那样简单应用的默认挂起的线程创建方式，因此我们自己设计了一个标志位，所有线程创建时进入while循环阻塞等待该标志位置True，相当于同时给多个线程发令，这时候的时间统计就是准确的。具体的实现方法见以下代码。

线程内部：

1. // 等待发令
2. **while** (!thread\_begin) {}

线程外部：

1. // 给多个线程同时发令
2. gettimeofday(&startv, &startz);
3. thread\_begin = **true**;

计时的同时给各线程“发令起跑”。

### 求和

#### 非SSE加速

不加速的求和方法就是简单的循环+累加，为了延长计算的时间，累加的同时对数据的处理（取log10和sqrt()）。

1. // 不加速版本求和
2. **double** NewSum(**const** **double** data[], **const** **int** len)
3. {
4. **double** rlt = 0.0f;
6. **for** (**int** i = 0; i < len; i++) rlt += log10(sqrt(data[i]/4.0));
8. **return** rlt;
9. }

函数传入参数为待求和的数组和它的长度，计算后返回求和结果。

#### SSE加速

使用在原理部分“1.3. SSE指令集”中所说到的方法，对求和进行简单的加速，这里选择使用的是128位的SSE指令，因为数据的存储形式为双精度double型，所占存储空间为64位（8个字节），所以128位可以同时进行两组的运算，理论上可以将求和加速到原来的两倍。函数实现如下：

1. // SSE加速求和
2. **double** NewSumSSE(**const** **double**\* pbuf, **const** **int** len)
3. {
4. **double** sum = 0;
6. **int** nBlockWidth = 2;
7. **int** cntBlock = len/nBlockWidth;
8. **int** cntRem = len%nBlockWidth;
10. **const** **double**\* p = pbuf;
11. **const** **double**\* q;
13. \_\_m128d xfsload;
14. \_\_m128d xfssum = \_mm\_setzero\_pd();
16. **for** (**int** i = 0; i < cntBlock; i++)
17. {
18. xfsload = \_mm\_sqrt\_pd(\_mm\_sqrt\_pd(\_mm\_load\_pd(p)));
19. xfssum = \_mm\_add\_pd(xfssum, xfsload);
20. p += nBlockWidth;
21. }
23. q = (**const** **double**\*)&xfssum;
24. **for** (**int** i = 0; i < nBlockWidth ; i++)
25. {
26. sum += q[i];
27. }
29. // 提取剩余的
30. **for** (**int** i = 0; i < cntRem; i++)
31. {
32. sum += p[i];
33. }
35. **return** sum;
36. }

首先将128位分成2块，用以存放double，然后依次进行计算，将结果累加到一个128位的数据中。最后进行提取，并对剩余未作运算的数据累加入结果当中。在当前double的数据类型中，假如求和的长度为奇数，那么就会剩下一个数无法组合（不能保证下一个内存地址中的值是否刚好为0），需要直接提取累加。

### 求最大值

#### 作数据处理的求最大值

求最大值的方法也很简单，直接对待求数据进行一次遍历。为了增加消耗的时间，在遍历的同时也对数值作类似于求和部分的处理，存储其中最大值数值。

1. // 不加速版本求最大值
2. **double** NewMax(**const** **double** data[], **const** **int** len)
3. {
4. **double** max = log10(sqrt(data[0]/4.0));
6. **for** (**int** i = 1; i < len; i++) **if** (log10(sqrt(data[i]/4.0)) > max) max = log10(sqrt(data[i]/4.0));
8. **return** max;
9. }

函数传入参数为待求最大值的数据数组和长度，计算后返回最大值。

#### 单纯求最大值

由于在对多线程或多机的结果进行合并时，不需要再对数据边遍历边处理，所以使用以下函数求得最大值。

1. // 单纯求最大值（用于归并结果）
2. **double** NewMax\_2(**const** **double** data[], **const** **int** len)
3. {
4. **double** max = data[0];
6. **for** (**int** i = 1; i < len; i++) **if** (data[i] > max) max = data[i];
8. **return** max;
9. }

函数传入参数为待求最大值的数据数组和长度，计算后返回最大值。

### 排序

本次大作业编写了两种排序算法：冒泡法排序和快速排序。两者在应用到多机、多线程的时候有着截然不同的效果，具体的分析见后文的“4. 测试与结果分析”。算法的实现如下。

#### 冒泡法排序

1. **double** NewSort\_0(**const** **double** data[], **const** **int** len, **double** result[])
2. {
3. **double** tmp;
5. **for** (**int** i = 0; i < len; i++) result[i] = data[i];
7. **for** (**int** i = 0; i < len; i++)
8. {
9. **for** (**int** j = 0; j < len - i - 1; j++)
10. {
11. **if** (log10(sqrt(result[j])) > log10(sqrt(result[j + 1])))
12. {
13. tmp = result[j];
14. result[j] = result[j + 1];
15. result[j+1] = tmp;
16. }
17. }
18. }
19. }

就是在C语言基础课上熟悉掌握的冒泡法排序，时间复杂度为o(N2)，输入参数为待排序的数据和长度、结果存放的数组。

#### 快速排序

快速排序所采用的是一种“分而治之”的思想，将一个问题分成子问题，解决方法（算法）对于子问题同样适用，可以再分为子问题。算法实现如下。

1. // 快速排序算法
2. **void** qsort(**double** s[], **int** l, **int** r)
3. {
4. **if** (l < r)
5. {
6. **int** i = l, j = r;
7. **double** x = s[i];
8. **while** (i < j)
9. {
10. **while** (i < j && s[j] >= x)
11. {
12. j--;
13. }
14. **if** (i < j)
15. {
16. s[i++] = s[j];
17. }
18. **while** (i < j && s[i] < x)
19. {
20. i++;
21. }
22. **if** (i < j)
23. {
24. s[j--] = s[i];
25. }
26. }
27. s[i] = x;
28. qsort(s, l, i - 1);
29. qsort(s, i + 1, r);
30. }
31. }

首先将一串数据最左端作为参考，将i和j分别对应数据的最左端和最右端。j向左移，找到比参考值小的数字，放入i对应的位置，然后将i向右移，找到比参考值大的数字，放入j的位置。就这样不断循环直到i与j相遇，达到的效果是：左段均是比参考值小的数，右段均是比参考值大的数。此时左右两段仍是无序的，所以将左右段分别再进行同样的操作。最后排序完成。

由于要对应要求当中的函数接口，所以额外写了一段调用快速排序的函数。

1. // 不加速排序（快排）
2. **double** NewSort(**const** **double** data[], **const** **int** len, **double** result[])
3. {
4. **int** l = 0, r = len - 1;
5. **for** (**int** i = 0; i < len; i++)
6. {
7. result[i] = data[i];
8. }
9. qsort(result, l, r);
10. }

这样就能跟冒泡法排序保持一致性，方便替换。

#### 排序结果的检验

排序结果需要进行检验，采用的方法是前后差分，得到符号值（正负表示），当出现异号时，说明序列不是有序的。具体实现的方法如下。

1. // 检验结果
2. **int** check(**const** **double** data[], **const** **int** len)
3. {
4. **double** sign;
5. **double** new\_sign;
7. sign = data[1] - data[0];
8. **for** (**int** i = 1; i < len - 1; i++)
9. {
10. new\_sign = data[i + 1] - data[i];
11. **if** (new\_sign\*sign < 0)
12. {
13. printf("i = %d, %f, %f\r\n", i, data[i], data[i + 1]);
14. **return** 0;
15. }
16. }
18. **return** 1;
19. }

其中为了方便调试，当出现异号的时候，输出当前在序列当中的位置，并输出前后两数的数值。函数输入的参数为待检验的数组和长度，经过计算后返回检验结果，1为有序，0为无序。

## 单机单线程

单机单线程的编程流程十分简单，就是直接将生成的数据放入到函数中，完全遍历，所以耗时也将会是最长。这里以求和为例，其它运算类似。

1. // ------------------------ SUM begin ------------------------
2. finalSum = 0.0f;
3. gettimeofday(&startv, &startz);
4. finalSum = NewSum(rawDoubleData, DATANUM);
5. gettimeofday(&endv, &endz);
6. t\_usec = (endv.tv\_sec - startv.tv\_sec)\*1000000 + (endv.tv\_usec - startv.tv\_usec);
7. printf("Single-PC-single-thread[SUM](Server): answer = %lf, time = %ld us\r\n", finalSum, t\_usec);
8. // ------------------------ SUM end --------------------------

在计算前先记录当前的时间，计算完成后记录完成的时间，前后做差得到耗时大小。将计算结果和消耗时间输出以作对比。

## 单机多线程

单机多线程的编程同样只在此阐述求和的方法，其它的运算类似。

1. // 每个线程的ID
2. **int** id[MAX\_THREADS];
3. **for** (**int** i = 0; i < MAX\_THREADS; i++) id[i] = i;
5. // 多线程相关
6. pthread\_t tid[MAX\_THREADS];
7. pthread\_attr\_t attr;
8. **size\_t** stacksize;
10. // 更改栈的大小
11. pthread\_attr\_init(&attr);
12. pthread\_attr\_getstacksize(&attr, &stacksize);
13. stacksize \*= 4;
14. pthread\_attr\_setstacksize(&attr, stacksize);

首先需要给每个线程ID，然后初始化一些跟多线程相关的变量。注意将线程的堆栈大小变为原来的四倍，以满足对大量数据运算的需求。

1. // ------------------------ SUM begin ------------------------
2. thread\_begin = **false**;
3. **for** (**int** i = 0; i < MAX\_THREADS; i++) pthread\_create(&tid[i], &attr, fnThreadSum\_0, &id[i]);
5. // 给多个线程同时发令
6. gettimeofday(&startv, &startz);
7. thread\_begin = **true**;

将标志位先置为False，然后逐个线程创建并传入ID。需要开辟额外的内存空间来存放ID，而不是将此时的i传入给各线程的原因是：将i传入线程，线程得到的是i的内存地址，当它去查看该值时，主线程有可能已经继续往下遍历，改变了该内存地址上的值，导致线程获取的ID并不正确。有两种解决办法：

1. 在创建线程时，每创建一个线程后延时一小段时间，保证线程能读出正确的ID，但缺点是，耗时长，且为开环控制，在不同环境下难以保证其正确性。
2. 第二种方法就是现在采用的办法，将ID提前存好在各自的地址，传入参数的时候将该地址传入。没有任何操作会对该地址中的内容作更改，保证了其正确性。

随后，记录开始时间的同时，给线程“发令”，开始计算。

在线程的函数里，对原始数据进行分段获取后，调用函数进行求和运算。

1. **void**\* fnThreadSum\_0(**void** \*arg)
2. {
3. **int** who = \*(**int** \*)arg;    // 线程ID
4. **double** data[SUBDATANUM];  // 线程数据
6. // 索引
7. **int** startIndex = who\*SUBDATANUM;
8. **int** endIndex = startIndex + SUBDATANUM;
10. **for** (**int** i = startIndex, j = 0; i < endIndex; i++, j++) data[j] = rawDoubleData[i];
12. // 等待发令
13. **while** (!thread\_begin) {}
15. // 存储结果
16. doubleResults[who] = NewSum(data, SUBDATANUM);
18. **return** NULL;
19. }

可以看到，在线程函数里，获取前文所说的ID值，并得到本线程对应的数据段，对该数据求和，存放在所有线程各自结果数组，本线程所属的结果中。

1. // 等待线程运行结束
2. **for** (**int** i = 0; i < MAX\_THREADS; i++) pthread\_join(tid[i], NULL);
4. // 收割
5. finalSum = 0.0f;
6. **for** (**int** i = 0; i < MAX\_THREADS; i++) finalSum += doubleResults[i];
8. gettimeofday(&endv, &endz);
9. t\_usec = (endv.tv\_sec - startv.tv\_sec)\*1000000 + (endv.tv\_usec - startv.tv\_usec);
10. printf("Single-PC-multi-thread[SUM](Server): answer = %lf, time = %ld us\r\n", finalSum, t\_usec);
11. // ------------------------ SUM end --------------------------

等待所有线程结束后，对各线程的数据进行“收割”，得到最后的结果。并记录下结束的时间，与开始时间做差得出计算所耗费的时间。

## 多机多线程

本次大作业使用的双机协作，原理不变的情况下可以多机进行协作，进一步加快计算的速度。在这一部分，以排序作为讲解的例子。无论是多机还是单机，只要是将数据分段进行排序的运算，就会遇到最后结果归并的问题。各段虽然有序，但各段之间的关系是无序的，这种排序的结果实际上还不如快速排序的中间态。所以在测试中我们也可以看到相应的现象，这也是我们迫不得已又采用冒泡排序的原因。

先是进行网络的配置和连接，分为服务端和客户端两种不同的操作。

### 服务端

1. // Server, Client socket描述符
2. **int** server\_fd, client\_fd;
3. // My address, remote address
4. **struct** sockaddr\_in my\_addr, remote\_addr;
5. // 一些网络收发相关变量
6. **int** ret, recv\_len, send\_len, sin\_size;
8. // 设置地址
9. memset(&my\_addr, 0, **sizeof**(my\_addr));   // reset
10. my\_addr.sin\_family = AF\_INET;           // IPV4
11. my\_addr.sin\_addr.s\_addr = INADDR\_ANY;   // Local IP
12. my\_addr.sin\_port = htons(PORT);         // Port

先定义了一些网络通信将要用到的变量：socket描述符、地址、收发数据长度等等。因为是服务端，所以将地址绑定为本地的IP，同时指定了端口号和配置使用IPV4的地址。

1. **if** ((server\_fd = socket(PF\_INET, SOCK\_STREAM, 0)) < 0)
2. {
3. printf("create server\_fd failed...\r\n");
4. **return** -1;
5. }
7. **if** ((ret = bind(server\_fd,(**struct** sockaddr \*)&my\_addr,**sizeof**(my\_addr))) < 0)
8. {
9. printf("bind server\_fd failed...\r\n");
10. **return** -1;
11. }
13. // 等待连接
14. listen(server\_fd, 5);
16. // 只针对一个client的情况
17. sin\_size = **sizeof**(**struct** sockaddr\_in);
18. **if** ((client\_fd = accept(server\_fd, (**struct** sockaddr\*)&remote\_addr, (socklen\_t \*)&sin\_size)) < 0)
19. {
20. printf("accept connection failed...\r\n");
21. **return** -1;
22. }
23. printf("# Accept client %s\r\n", inet\_ntoa(remote\_addr.sin\_addr));

按照原理所述，依次绑定套接字、地址，并打开网络监听等待连接。当有连接时，将其与客户端的socket描述符进行绑定，同时得到客户端的地址。在这一部分，此次大作业只针对一个客户端的情况。

1. // 对Client的应答
2. memset(buf, 0, BUF\_LEN);
3. sprintf(buf, "Hello Client!");
4. **if** ((send\_len = send(client\_fd, buf, strlen(buf), 0)) < 0)
5. {
6. printf("server send failed...\r\n");
7. **return** -1;
8. }
9. printf("# Send: Hello Client!\r\n");

发送一条消息，在客户端可查看到该消息，验证双方连接的连通性。

### 客户端

客户端的编程相比服务端要比较简单。它不需要绑定自己的地址，也不需要打开监听。只需要配置已知的服务端IP地址和端口号，就可以尝试连接。具体的实现如下。

1. // Client socket描述符
2. **int** client\_fd;
3. // My address, remote address
4. **struct** sockaddr\_in remote\_addr;
5. // 一些网络收发相关变量
6. **int** ret, recv\_len, send\_len;
8. // 设置地址
9. memset(&remote\_addr,0,**sizeof**(remote\_addr));         // reset
10. remote\_addr.sin\_family = AF\_INET;                   // IPV4
11. remote\_addr.sin\_addr.s\_addr = inet\_addr(SERVER\_IP); // Server IP
12. remote\_addr.sin\_port = htons(PORT);                 // Port

同样地，定义各种变量，使用IPV4的地址格式，将IP地址绑定为已知的服务端IP地址，并指定端口号。

1. **if** ((client\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) < 0)
2. {
3. printf("create client\_fd failed...\r\n");
4. **return** -1;
5. }
7. **if** ((ret = connect(client\_fd, (**struct** sockaddr\*)&remote\_addr, **sizeof**(remote\_addr))) < 0)
8. {
9. printf("connect failed...\r\n");
10. **return** -1;
11. }
12. printf("Connect to server!\r\n");
14. **if** ((recv\_len = recv(client\_fd, buf, BUF\_LEN, 0)) < 0)
15. {
16. printf("client recv failed...\r\n");
17. **return** -1;
18. }
19. printf("# Received: %s\r\n", buf);

绑定套接字，然后就可以尝试连接前面定义好的地址。当连接成功后，等待接收前文服务端发送的验证消息，验证双方连接的连通性。

### 多线程

随后就是双机各自开启多线程运算，这里就不再赘述，与前文“单机多线程”部分基本一致。在此将讲解我们是如何归并多线程排序的结果和多机排序的结果。

1. **void** merge()
2. {
3. **int** index[MAX\_THREADS];
4. **for** (**int** i = 0; i < MAX\_THREADS; i++)
5. {
6. index[i] = 0;
7. }
8. **int** min\_index;
9. **for** (**int** j = 0; j < S\_CLT\_DATANUM; j++)
10. {
11. **double** min\_num = log10(sqrt(DATA\_MAX));
12. **for** (**int** i = 0; i < MAX\_THREADS; i++)
13. {
14. **if** (index[i] > S\_CLT\_SUBDATANUM - 1)
15. {
16. **continue**;
17. }
19. **if** (S\_threadResult[i][index[i]] < min\_num)
20. {
21. min\_index = i;
22. min\_num = S\_threadResult[i][index[i]];
23. }
24. }
25. S\_CLT\_sortDoubleData[j] = min\_num;
26. index[min\_index]++;
27. }
28. }

假设有64线程，在多线程得到64个排序结果后，每个结果内部都是有序的，所以我们只需要从每个结果的第0位开始查找，就能找到最终结果中最小的（各线程的排序方法是从小到大排序）数值。依此类推，每得到一个数值就要对64个数求最小值，这种运算量不容小觑。

同样的思路，将双机的结果归并，可以得到最后的排序结果。由于数据量太大时，网络传输有可能会发生丢包错误。只要发生一次错误，排序结果就会出错，这也是一直困扰我们的问题。同时，使用这种归并方法，并不能使多线程发挥效用，反而单线程的快速排序具有更小的时间复杂度(o(NlgN))。

# 测试与结果分析

## 测试环境

操作系统：server, client均为Linux, Ubuntu 16.04 LTS；

处理器：server为Intel(R) Core(TM) i5-8300H CPU 2.3GHz，4核；

client为Intel(R) Core(TM) i5-8250U CPU 1.6GHz，4核；

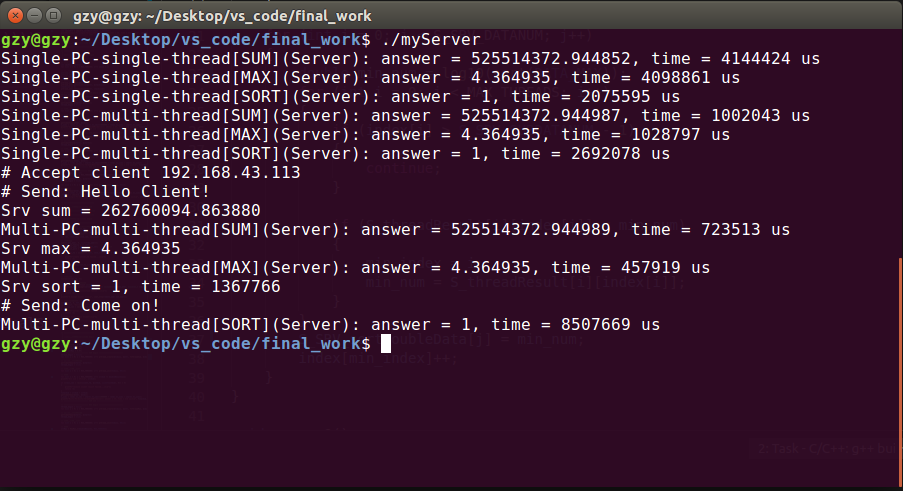
均不支持超线程。

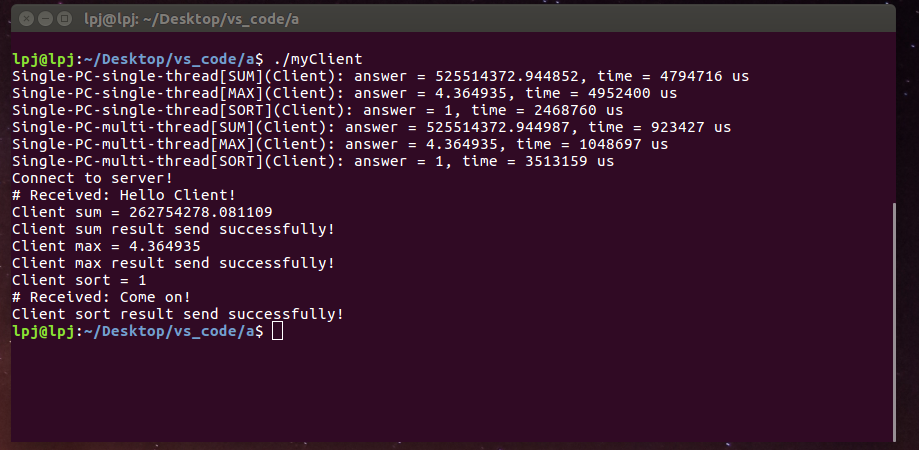
编译器：gcc 5.4.0。

## 测试结果

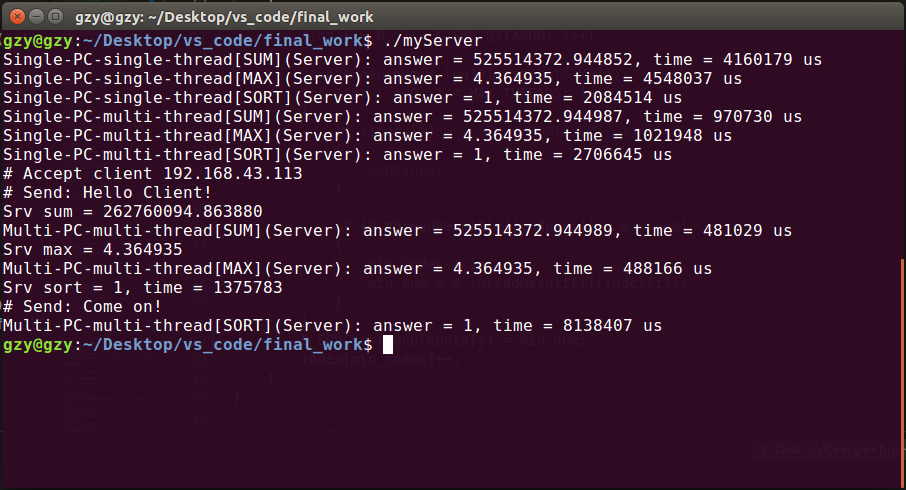
测试结果的截图附上，将在后文以表格的形式加以分析。

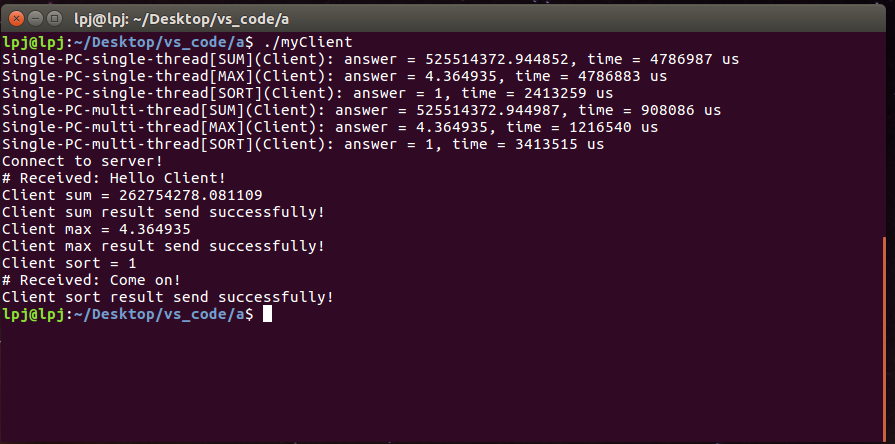
### 测试1



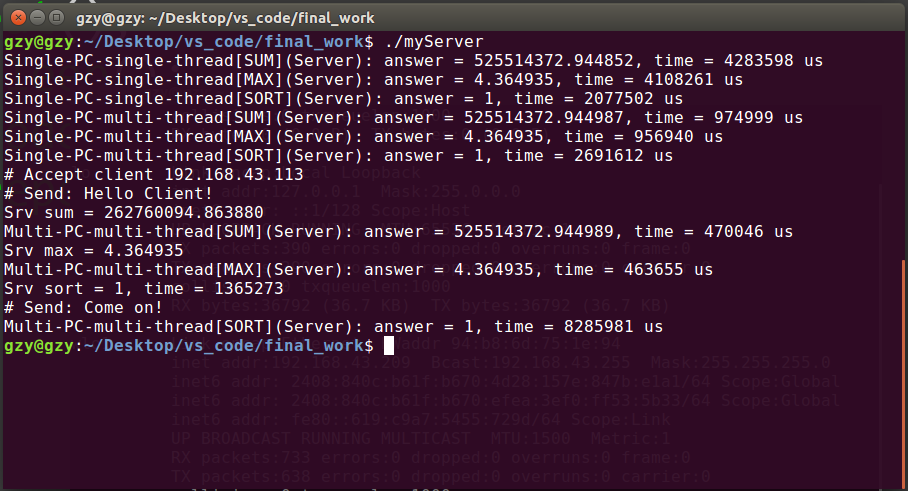


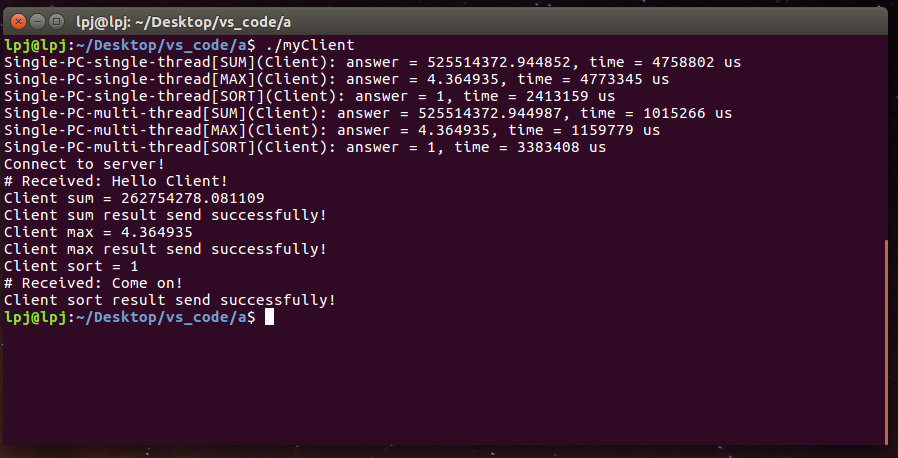
### 测试2



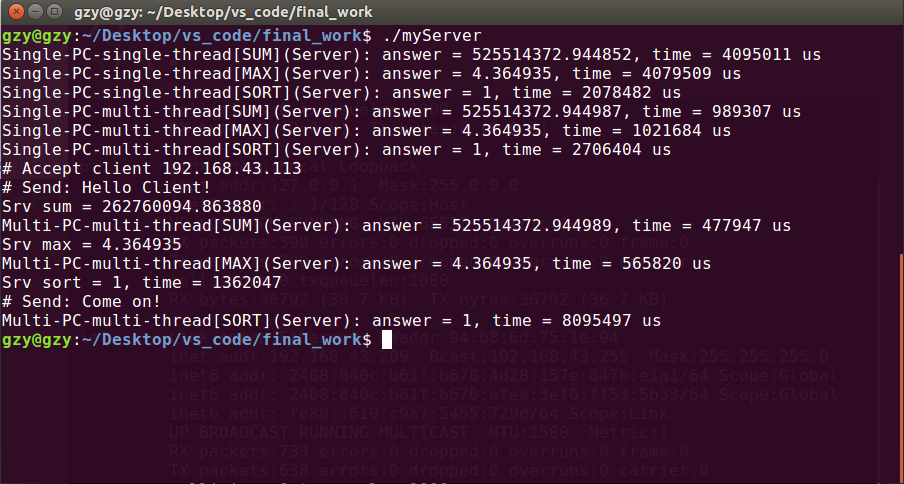


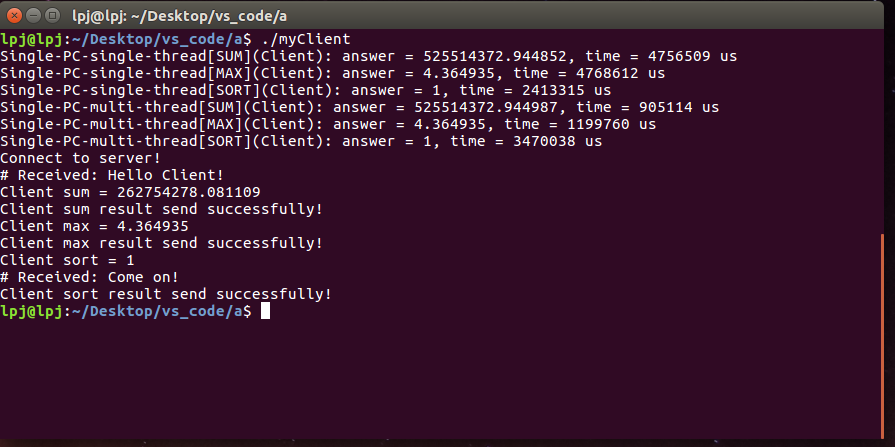
### 测试3



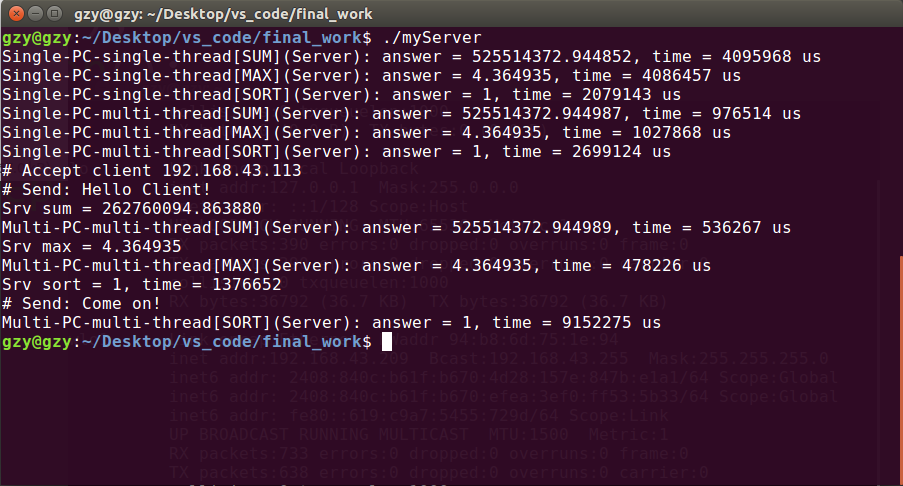


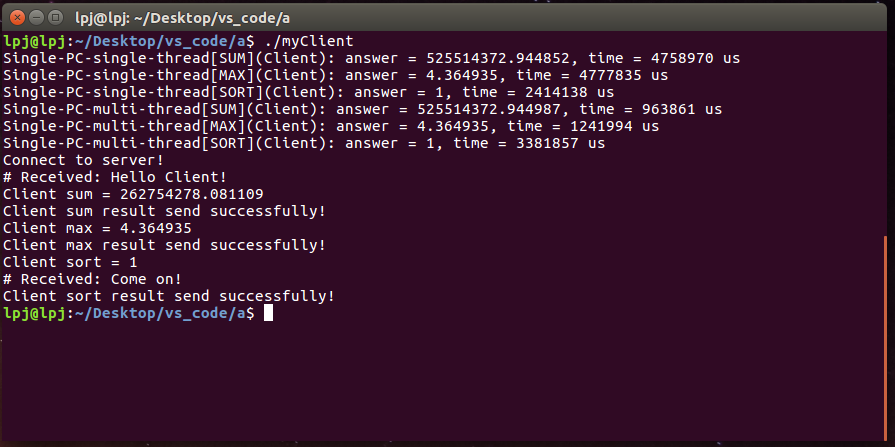
### 测试4





### 测试5





### 求和结果

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 序号 | A单机单线程SUM | B单机单线程SUM | 平均单机单线程SUM | A单机多线程SUM | B单机多线程SUM | 平均单机多线程SUM | 多机多线程SUM |
| ~~1~~ | ~~4144424~~ | ~~4794716~~ | ~~4469570~~ | ~~1002043~~ | ~~923427~~ | ~~962735~~ | ~~723513~~ |
| 2 | 4160179 | 4786987 | 4473583 | 970730 | 908086 | 939408 | 481029 |
| 3 | 4283598 | 4758802 | 4521200 | 974999 | 1015266 | 995132.5 | 470046 |
| 4 | 4095011 | 4756509 | 4425760 | 989307 | 905114 | 947210.5 | 477947 |
| ~~5~~ | ~~4095968~~ | ~~4758970~~ | ~~4427469~~ | ~~976514~~ | ~~963861~~ | ~~970187.5~~ | ~~536267~~ |
| 平均 |  |  | 4473514.333 |  |  | 960583.6667 | 476340.6667 |
| 加速比 |  |  |  |  |  | 4.657079324 | 9.391418047 |

时间单位均为us，因为有两组数据远远偏离于其他数据，因此将其剔除。求得求和的单机单线程平均时间为4473514.333us，单机多线程平均时间为960583.6667us，多机多线程平均时间为476340.6667us。得出求和时，相对于单机单线程而言，单机多线程的加速比为4.7，多机多线程加速比为9.4。

### 求最大值结果

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 序号 | A单机单线程MAX | B单机单线程MAX | 平均单机单线程MAX | A单机多线程MAX | B单机多线程MAX | 平均单机多线程MAX | 多机多线程MAX |
| 1 | 4098861 | 4952400 | 4525630.5 | 1028797 | 1048697 | 1038747 | 457919 |
| 2 | 4548037 | 4786883 | 4667460 | 1021948 | 1216540 | 1119244 | 488166 |
| 3 | 4108261 | 4773345 | 4440803 | 956940 | 1159779 | 1058359.5 | 463655 |
| ~~4~~ | ~~4079509~~ | ~~4768612~~ | ~~4424060.5~~ | ~~1021684~~ | ~~1199760~~ | ~~1110722~~ | ~~565820~~ |
| 5 | 4086457 | 4777835 | 4432146 | 1027868 | 1241994 | 1134931 | 478226 |
| 平均 |  |  | 4516509.875 |  |  | 1087820.375 | 471991.5 |
| 加速比 |  |  |  |  |  | 4.151889392 | 9.569049178 |

时间单位均为us，因为有一组数据远远偏离于其他数据，因此将其剔除。求得求最大值的单机单线程平均时间为4516509.875us，单机多线程平均时间为1087820.375us，多机多线程平均时间为471991.5us。得出求最大值时，相对于单机单线程而言，单机多线程的加速比为4.2，多机多线程加速比为9.6。

### 排序结果

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 序号 | A单机单线程SORT | B单机单线程SORT | 平均单机单线程SORT | A单机多线程SORT | B单机多线程SORT | 平均单机多线程SORT | 多机多线程SORT |
| 1 | 2075595 | 2468760 | 2272177.5 | 2692078 | 3513159 | 3102618.5 | 8507669 |
| 2 | 2084514 | 2413259 | 2248886.5 | 2706645 | 3413515 | 3060080 | 8138407 |
| 3 | 2077502 | 2413159 | 2245330.5 | 2691612 | 3383408 | 3037510 | 8285981 |
| 4 | 2078482 | 2413315 | 2245898.5 | 2706404 | 3470038 | 3088221 | 8095497 |
| ~~5~~ | ~~2079143~~ | ~~2414138~~ | ~~2246640.5~~ | ~~2699124~~ | ~~3381857~~ | ~~3040490.5~~ | ~~9152275~~ |
| 平均 |  |  | 2253073.25 |  |  | 3072107.375 | 8256888.5 |
| 加速比 |  |  |  |  |  | 0.733396648 | 0.272871948 |

时间单位均为us，因为有一组数据远远偏离于其他数据，因此将其剔除。求得排序的单机单线程平均时间为2253073.25us，单机多线程平均时间为3072107.375us，多机多线程平均时间为8256888.5us。得出排序时，相对于单机单线程而言，单机多线程的加速比为0.73，多机多线程的加速比为0.27。

## 结果分析

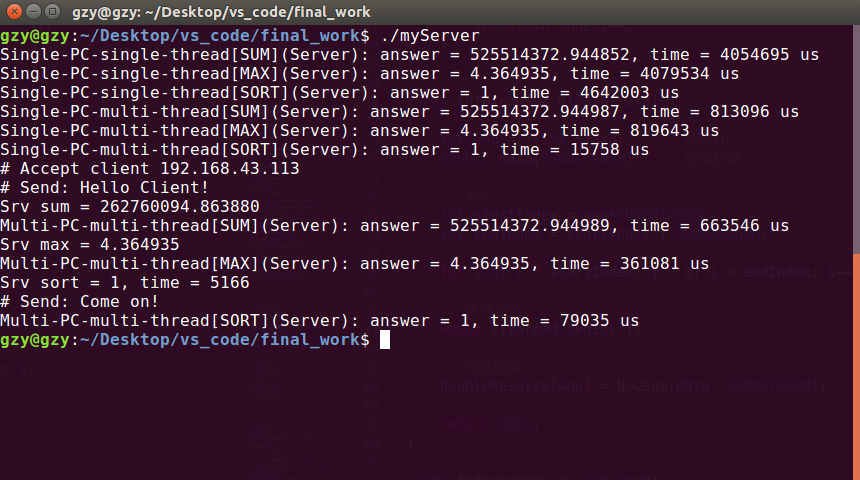
分析求和与求最大值的结果，可以看到单机多线程达到了4倍多的加速比，与4核CPU的配置预期结果相符。而在多机多线程的时候达到了9倍多的加速比，也符合预期。

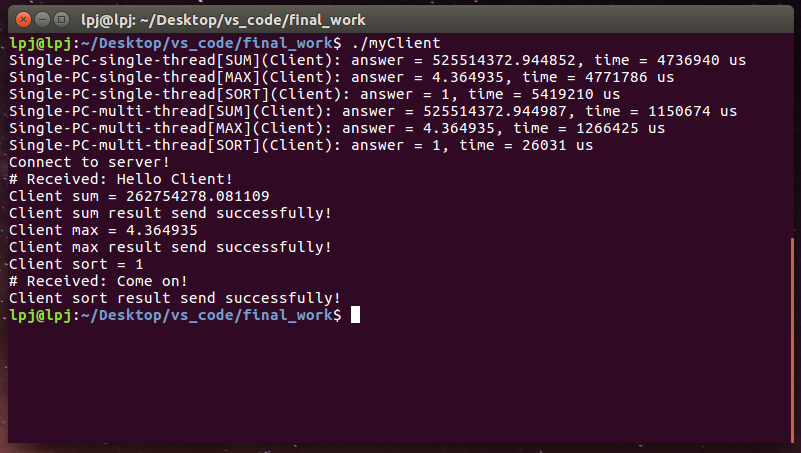
但是在进行排序的测试时，单机多线程与多机多线程并没有加速效果，反而计算得更慢。这也是由于我们使用了快速排序，对每一块数据进行了分块，但是由于归并的算法比较简单，耗时比较长且使用的是单线程的归并方法，所以对结果的归并消耗了更多的时间。考虑到快速排序的时间复杂度为o(NlgN)，在大量数据的时候有显著的优势，因此又选用了时间复杂度为o(N2)的冒泡法排序进行测试，由于冒泡法排序对于大量数据的处理实在是无能为力，所以减少分块的数据量和数据总量，采用64×2000的数据规模进行跟上文一样的测试。

## 附加测试与结果分析

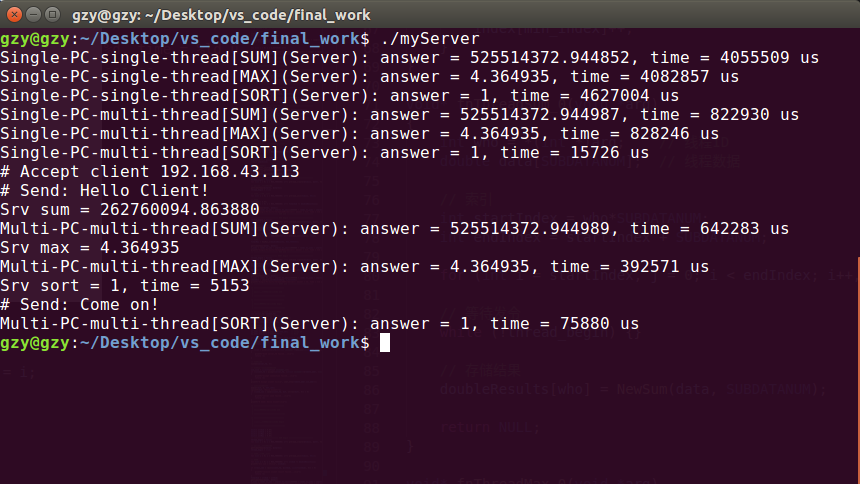
实验的结果截图如下。

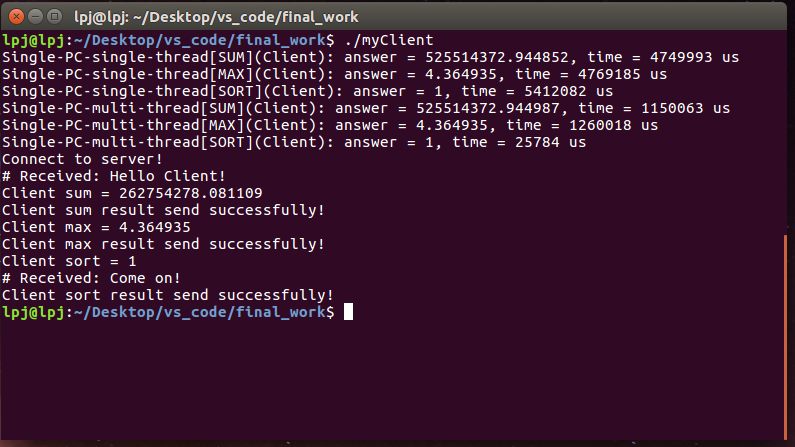
### 测试1



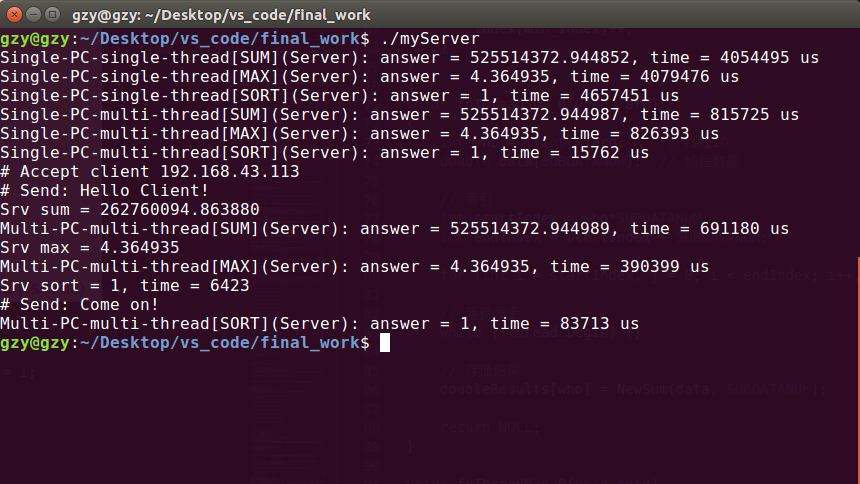


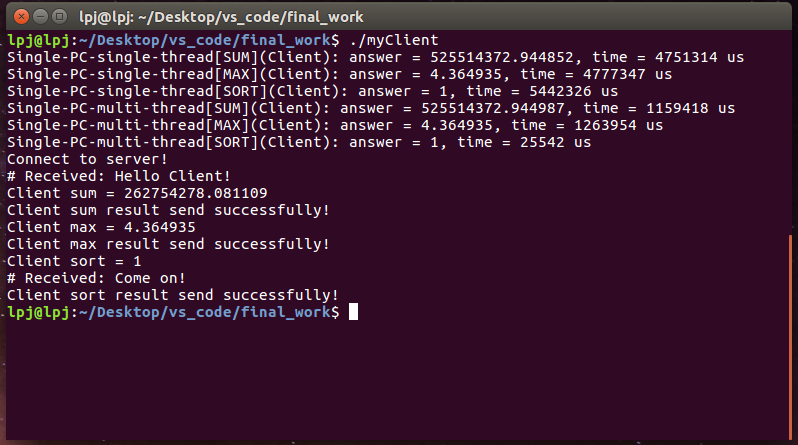
### 测试2



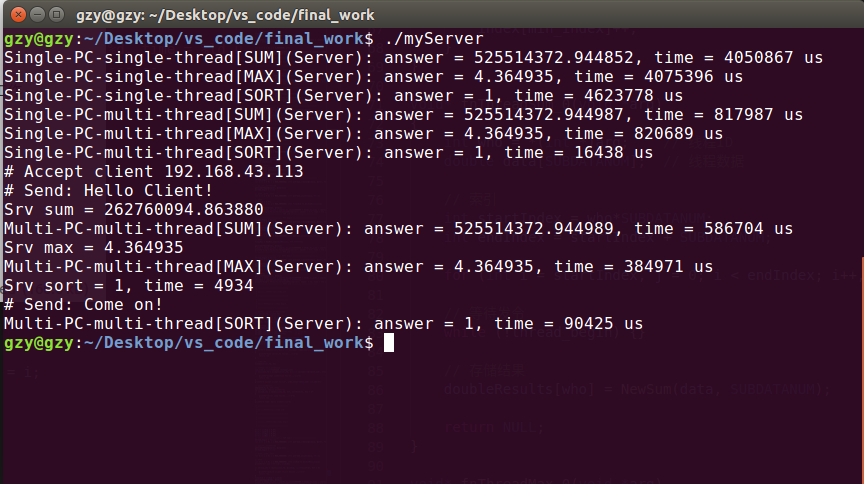


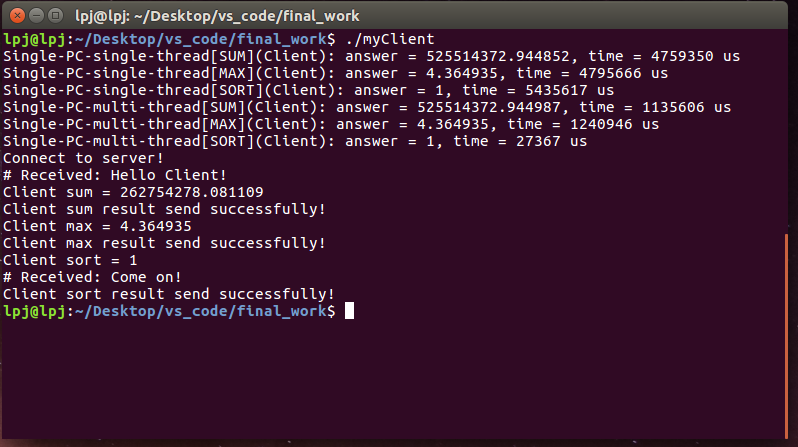
### 测试3



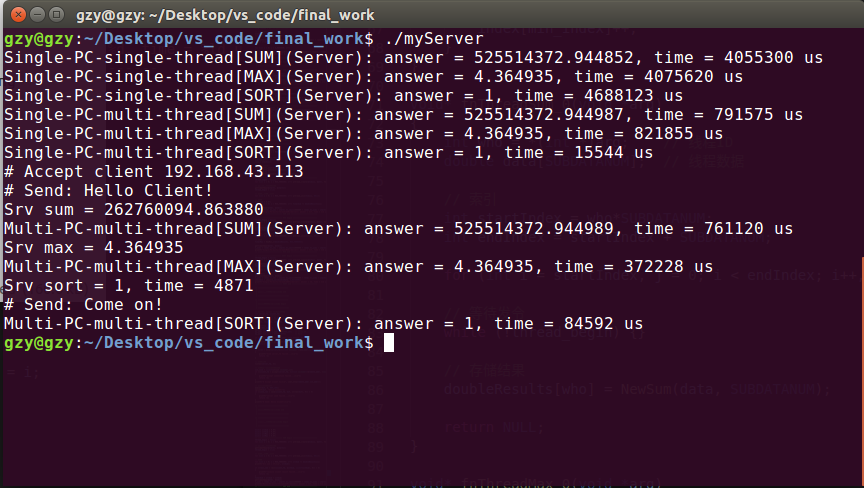


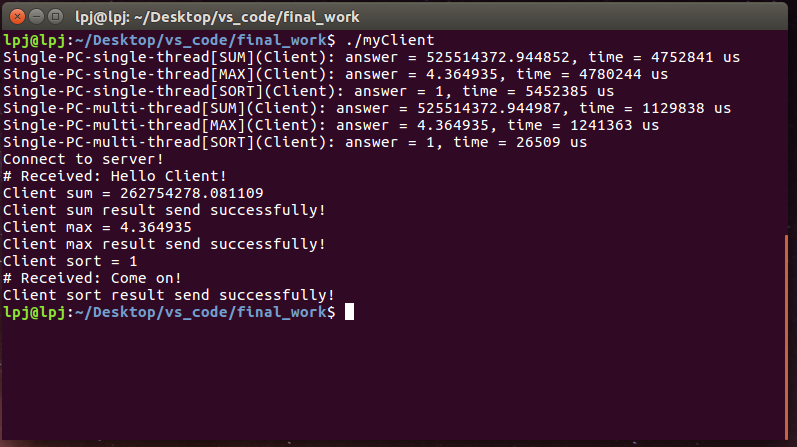
### 测试4





### 测试5





### 结果与分析

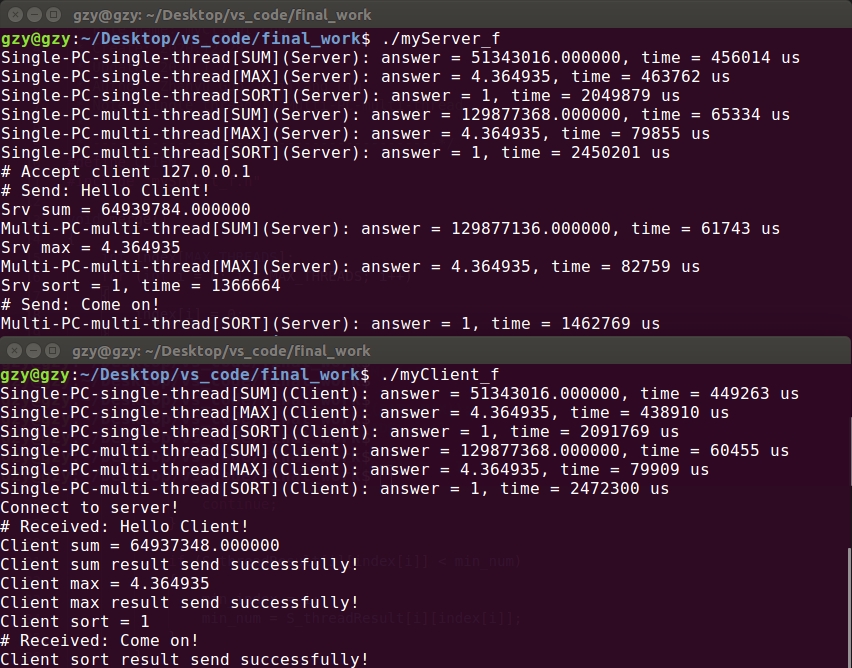
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 序号 | A单机单线程SORT | B单机单线程SORT | 平均单机单线程SORT | A单机多线程SORT | B单机多线程SORT | 平均单机多线程SORT | 多机多线程SORT |
| 1 | 4642003 | 5419210 | 5030606.5 | 15758 | 26031 | 20894.5 | 79035 |
| 2 | 4627004 | 5412082 | 5019543 | 15726 | 25784 | 20755 | 75880 |
| 3 | 4657451 | 5442326 | 5049888.5 | 15762 | 25542 | 20652 | 83713 |
| 4 | 4623778 | 5435617 | 5029697.5 | 16438 | 27367 | 21902.5 | 90425 |
| 5 | 4688123 | 5452385 | 5070254 | 15544 | 26509 | 21026.5 | 84592 |
| 平均 |  |  | 5039997.9 |  |  | 21046.1 | 82729 |
| 加速比 |  |  |  |  |  | 239.4741971 | 60.92177955 |

时间单位均为us。更换为计算效率不那么高的冒泡法排序后，求得排序的单机单线程平均时间为5039997.9us，单机多线程平均时间为21046.1us，多机多线程平均时间为82729us。得出排序时，相对于单机单线程而言，单机多线程的加速比为239，多机多线程的加速比为61。

分析原因，是因为多线程进行排序的时候，大大减小了计算量。使用冒泡法排序64×200个数据，N1 = 12800，需要遍历N12 = 1.64e+8次；每个线程N2 = 200，需要遍历N22 = 4e+4，64个线程一共需要2.56e+6次，如果忽略归并时间，那么加速了1.64e+8/2.56e+6 = 64倍，因为多线程时有4个CPU同时工作，所以加速比为64×4 = 256。由于归并需要时间，所以并没有达到该速度，但239.5的加速比符合预期。在多机多线程运行时，虽然每个线程的数据量进一步减少，但是耗费的时间需要额外加上网络传输结果的时间，传输数据量为，64×100×8 = 50kBytes，同时还要对多机结果进行归并，所以加速效果不如单机多线程。

### SSE补充

因为项目临近完成时，仍未找到办法使用log10针对无论是float还是double类型进行计算（可能是Linux上的库版本落后），经过更新也无济于事。但在最终，群里有同学分享了软件算法实现128位单精度浮点数SSE计算，于是在此进行补充测试，同时验证了SSE的加速效果。



此次测试运行没有进行双机协作，而是单机同时运行服务端和客户端，理论上双机协作还能使速度提升为原来的1~2倍。可以看到，多线程运算消耗为60455us，而单线程为449263us，拥有的加速比为7.43，远高于未使用SSE的求和运算。理论上双机协作可以达到13以上的加速比。但同时也存在问题，由于使用的是单精度浮点数，求和的结果存在很大的误差，但运算精度并不是本次大作业的主要目的。

# 总结

## 课程设计总结

在本次课程设计中，我们的成果有：

（1）使用socket以及TCP协议实现多机之间的通信；

（2）使用多线程、SSE指令集以及显卡（效果不好没用上）等方式实现计算加速；

（3）使用linux系统实现目标。

通过本次课程设计，我们的收获有：

（1）我们对linux系统编程有了更深刻的了解、切实提高了编程能力；

（2）强化了小组合作精神，通过分工协作完成了复杂项目；

（3）对课本中所讲述的多机通信、多线程等知识有了更深刻的体会，不再是仅仅浮于概念和理论，而是切身编程一一实现了他们。

经历本次课程，我们的感悟有：

（1）学习计算机一定不能浮于理论和表面，而是一定要亲自动手编程才能更好地记忆和理解计算机的原理；

（2）在完成一个项目之前，一定要做好时间规划。例如，在某一个时间之前得完成某一项任务。不然的话，很有可能会出现手忙脚乱、顾此失彼的情况；

（3）在完成一个模块之前，可以先从简单的做起。例如，在多机通信模块，可以先试着传输单个数据检验框架的正确性。然后再逐步拓展成目标模块。

## 课程总结

一学期的课程眨眼间就结束了，整个学期最大的收获可能就是学会了如何“加速”吧。王老师教会了我们如何榨取CPU资源、如何使用更加高效的指令集以及如何调用多台计算机联合工作。现在火爆的“云计算”、“大数据”不也就是建立在这些理论的基础之上吗？师傅领进门，修行在个人。可以说王老师真正带我们接触了这些前沿技术的基础，以后能不能为这个领域奉献自己的力量还取决于我们自己的努力。

而且我们也真正在其他课的学习中用到了这门课的知识。例如，在人工智能课程的五子棋设计中，我们在客户端就使用了多线程来接受来自服务端的棋盘消息，这样就代替了主线程中使用定时器来定时接受棋盘消息的方式，提高了程序的效率和运行的稳定性。

这门课的优点明显，因为王老师是真的会带着我们去做东西、去写代码的。我觉得如果换成其他老师来开这门课，很可能只是上课带着我们把PPT念一遍然后什么例程也不给就开始布置大作业，一学期下来什么也没学到。我觉得其他老师也可以参考王老师的方式，就是给例程、讲例程以及带着我们做例程。如果老师上课只讲一些假大空的东西，然后什么都要依靠我们自己学，那么我们来上大学的目的是什么呢？不会真的有人认为是培养自学能力吧，不会吧，不会吧。

最后，给王老师打个call！王老师牛逼！

# 重现时注意

## 编译

因为是在Linux上进行开发，所以编译有点讲究。我们使用的是vscode中的C/C++插件，在vscode中进行配置，生成对应的task.json文件，使用的gcc版本为5.4.0。task.json文件也会一并上传，而在里面的arg中，与默认不同的是，需要增加两句参数：

1. "args": [
2. "-g",
3. "${file}",
4. "-o",
5. "${fileDirname}/${fileBasenameNoExtension}",
6. "-lpthread",
7. "-mcmodel=large"
8. ],

也可以在git上直接下载到我们的代码：

<https://gitee.com/guo_zhanyu/multi-pc-multi-thread>

1. $ cd xxx
2. $ code .

打开vscode进行编译。

或直接gcc编译。

## IP地址更改

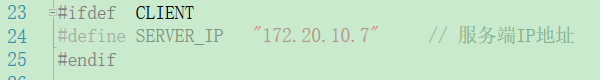
### IP地址存放所属文件

IP地址存放于头文件mySrvClt.h中，如图所示：



### IP地址修改位置

IP地址的修改位置位于mySrvClt.h头文件的第24行的位置，如图所示：



### IP地址修改说明

在Ubuntu系统查询IP地址时，在服务端电脑使用ifconfig命令查询服务端电脑的IP地址，然后进行修改。

## 运算数据量更改

### 运算数据量存放位置

运算数据量存放于头文件mySrvClt.h中，如图所示：



### 运算功能数据量修改位置

运算数据量的修改位置位于mySrvClt.h头文件的第28行至第32行的位置的位置，如图所示：



### 运算数据量修改说明

|  |  |  |  |
| --- | --- | --- | --- |
| 变量名 | 默认值 | 取值范围 | 备注 |
| MAX\_THREADS | 64 | [1, 61594] | 线程数 |
| SUBDATANUM | 2000000 | [1, 2000000] | 每个线程的子块数据量 |
| SRV\_SUBDATANUM | 1000000 | [1, 1000000] | 服务端每个线程的的子块数据量 |
| CLT\_SUBDATANUM | 1000000 | [1, 1000000] | 客户端每个线程的的子块数据量，两者之和要等于SUBDATANUM。 |
| - | - | - |
| DATANUM | - | - | (SUBDATANUM\*MAX\_THREADS) |

## 排序功能数据量更改

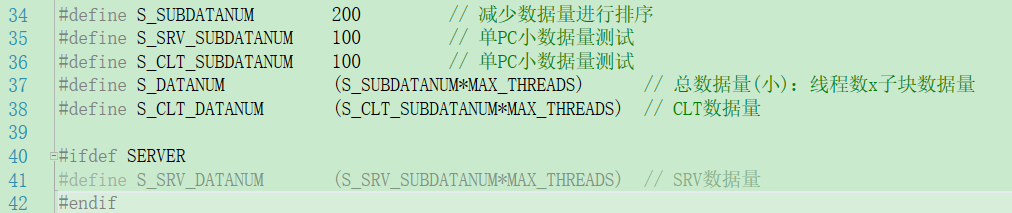
### 排序功能数据量存放位置

冒泡排序功能数据量存放于头文件mySrvClt.h中，如图所示：



### 排序功能数据量修改位置

排序功能数据量的修改位置位于mySrvClt.h头文件的第34行至第42行的位置的位置，如图所示：



### 排序功能数据量修改说明

|  |  |  |  |
| --- | --- | --- | --- |
| 变量名 | 默认值 | 取值范围 | 备注 |
| S\_SUBDATANUM | 200 | [1, 200000] | 排序时每个线程的子块数据量 |
| S\_SRV\_SUBDATANUM | 100 | [1, 100000] | 排序时服务端每个线程的的子块数据量 |
| S\_CLT\_SUBDATANUM | 100 | [1, 100000] | 排序时客户端每个线程的的子块数据量 |
| S\_DATANUM | - | - | (S\_CLT\_SUBDATANUM\*MAX\_THREADS) |
| S\_SRV\_DATANUM | - | - | (S\_SRV\_SUBDATANUM\*MAX\_THREADS) |
| S\_CLT\_DATANUM | - | - | (S\_CLT\_SUBDATANUM\*MAX\_THREADS) |

由于冒泡法排序单线程速度慢，但是快速排序可以接受更大的数据量，所以在采用冒泡法排序时，一般选择默认值附近的值利于体现效果。

## socket传输数据量更改

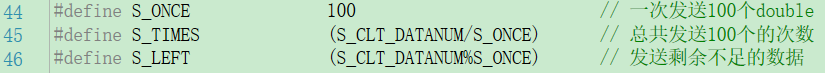
### socket传输数据量存放位置

socket传输数据量存放于头文件mySrvClt.h中，如图所示：



### socket传输数据量修改位置

socket传输数据量的修改位置位于mySrvClt.h头文件的第44行至第46行的位置的位置，如图所示：



### socket传输数据量修改说明

|  |  |  |  |
| --- | --- | --- | --- |
| 变量名 | 默认值 | 取值范围 | 备注 |
| S\_ONCE | 100 | [1, 180] | 排序时每个线程的子块数据量 |
| S\_TIMES |  |  | (S\_CLT\_DATANUM/S\_ONCE) |
| S\_LEFT |  |  | (S\_CLT\_DATANUM%S\_ONCE) |

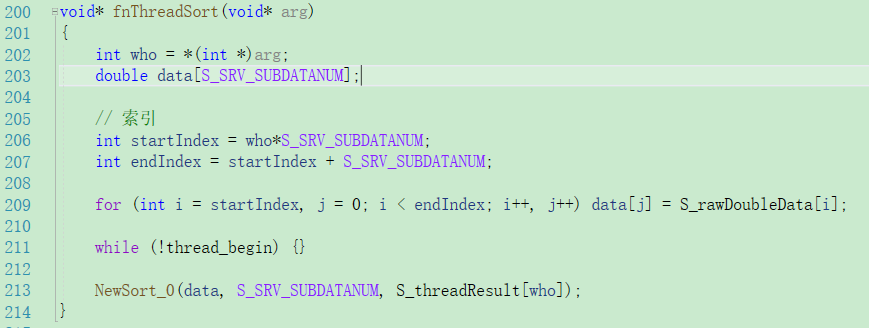
### 排序算法存放位置

排序算法存放于头文件mySrvClt.h中，如图所示：



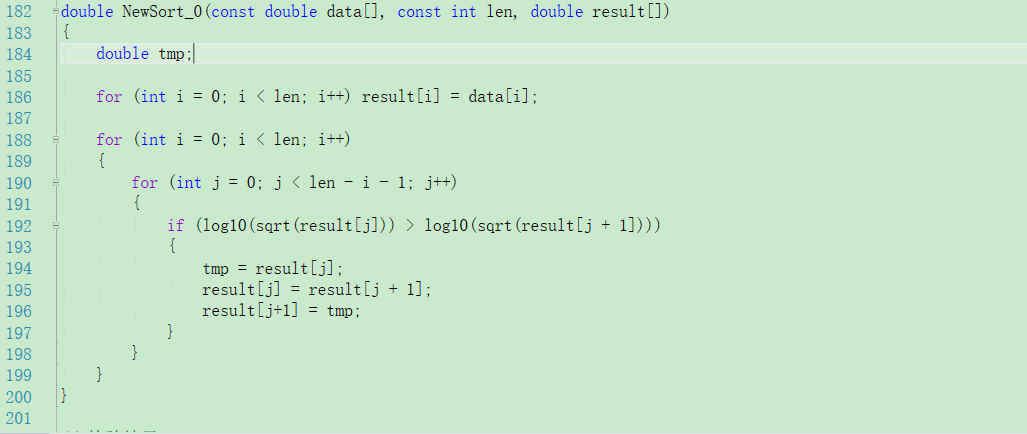
### 服务端排序算法的选择修改位置

服务端排序算法的选择修改位置位于myServer.cpp文件的第213行的位置的位置，大致如图所示：

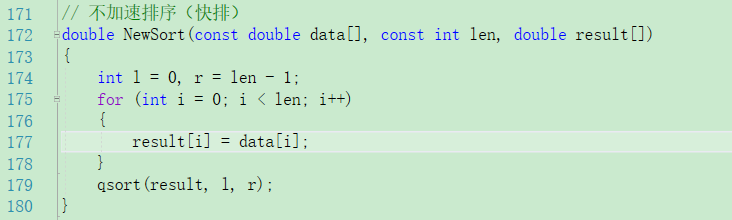


### 服务端排序算法的选择修改说明

1. 如果在第213行选择NewSort\_0函数，代表选择冒泡排序法，冒泡排序法如图所示：

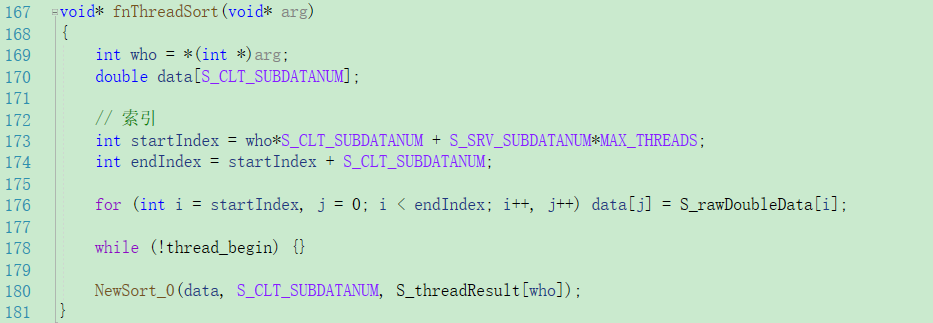


（2）如果在第213行选择NewSort函数，代表选择快速排序法，快速排序法如图所示：



### 客户端排序算法的选择修改位置

客户端排序算法的选择修改位置位于myClient.cpp文件的第180行的位置的位置，大致如图所示：



### 客户端排序算法的选择修改说明

|  |  |
| --- | --- |
| 函数名 | 备注 |
| NewSort\_0 | 冒泡排序法 |
| NewSort | 快速排序法 |

# 分工

|  |  |
| --- | --- |
| 工作 | 完成人 |
| 多机通信 | 郭展羽 |
| 多线程 | 郭展羽 |
| SSE加速 | 郭展羽、刘沛江 |
| 显卡加速（效果不好） | 刘沛江 |
| 报告代码实现部分、测试部分 | 郭展羽 |
| 其余 | 刘沛江 |

# 参考文献

[1] Richard John Anthony. 系统编程，分布式系统应用的设计与开发 [M]. 机械工业出版社, 2017.

[2] 张晓娜,常乐冉,吴炜,廖进蔚,沈立文. Linux系统下Socket通信的实现[J].电声技术, 2020, 44(01):87-89.

[3] 徐逸夫．Linux下基于socket多线程并发通信的实现[J]．通讯世界，2016(16):86．

[4] 周建国，晏蒲柳，郭成城．Linux下Client/Server异步通信的研究及实现[J]．计算机应用研究，2002(11):112-114.

[5] Nana Zhu, Hongyan Zhao [IoT applications in the ecological industry chain from information security and smart city perspectives](https://schlr.cnki.net/Detail/index/SJESLAST/SJES2125112A244A4BC0DC0C805EE959BE4A)[J]  Computers and Electrical Engineering, 2018, 65.