



同濟大學
TONGJI UNIVERSITY

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
1.1. 多线程.....	3
1.2. 多机通信与并行.....	5
1.3. SSE 指令集.....	7
2. 架构设计.....	9
3. 代码实现.....	11
3.1. 公用部分.....	11
3.2. 单机单线程.....	18
3.3. 单机多线程.....	19
3.4. 多机多线程.....	21
4. 测试与结果分析.....	25
4.1. 测试环境.....	25
4.2. 测试结果.....	25
4.3. 结果分析.....	31
4.4. 附加测试与结果分析.....	31
5. 总结.....	38
5.1. 课程设计总结.....	38
5.2. 课程总结.....	39
6. 重现时注意.....	40
6.1. 编译.....	40
6.2. IP 地址更改.....	40
6.3. 运算数据量更改.....	41
6.4. 排序功能数据量更改.....	41
6.5. socket 传输数据量更改.....	42
7. 分工.....	44
8. 参考文献.....	44

1. 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。

1.1. 多线程

1.1.1. 多线程及相关概念简介

1.1.1.1. 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:

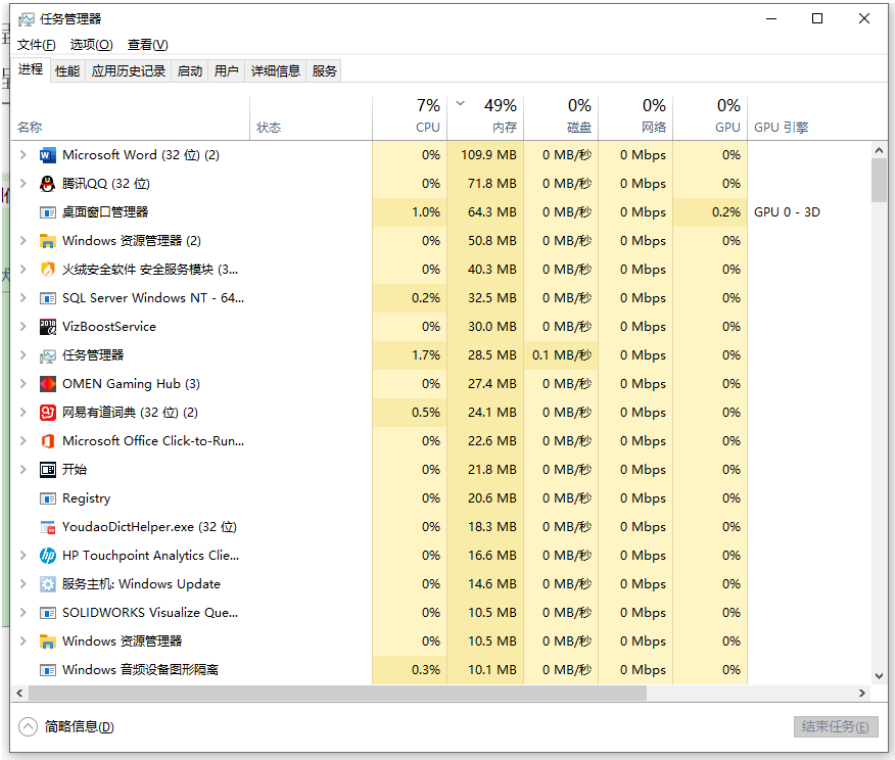


图 1-1 计算机上的进程

1.1.1.2. 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:

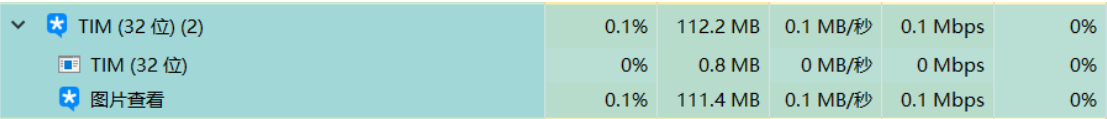


图 1-2 一个进程的多个线程

1.1.1.3. 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。

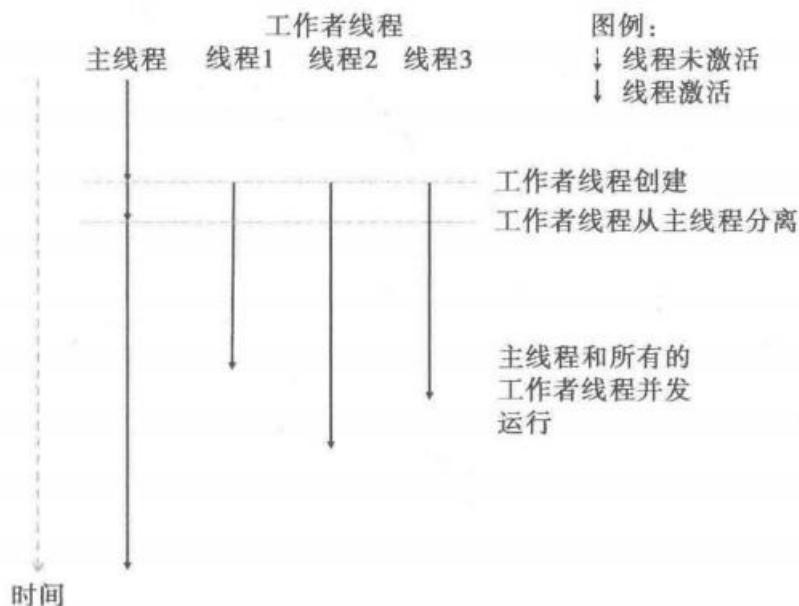


图 1-3 多线程例子

1.1.2. 多线程加速原理

1.1.2.1. 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。

1.1.2.2. 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。

1.1.2.3. 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：

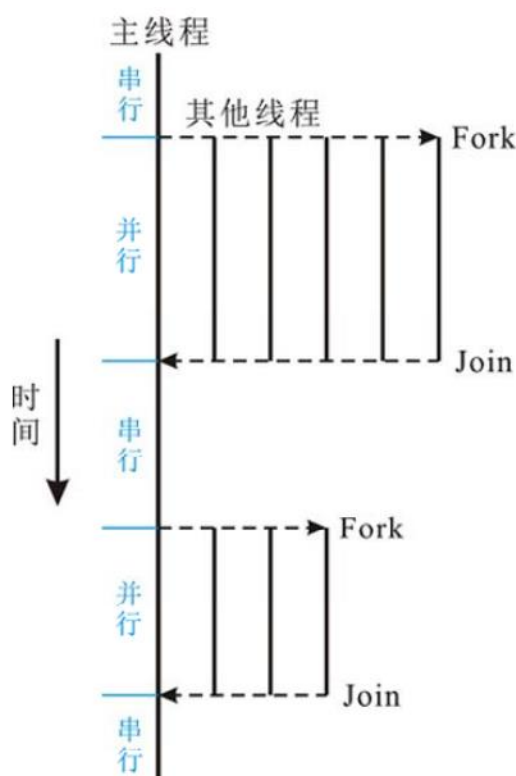


图 1-4 主线程与其他线程的关系

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。

1.2. 多机通信与并行

1.2.1. 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。

1.2.2. 多机通信

1.2.2.1. 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。

1.2.2.2. 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。

1.2.2.3. 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：

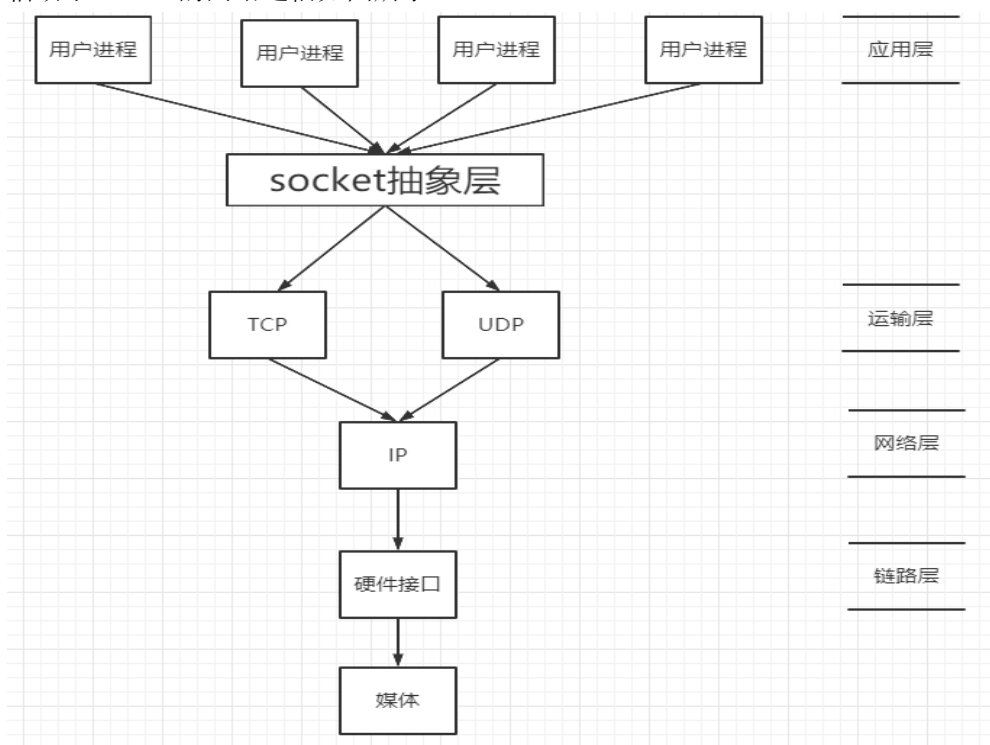


图 1-5 socket 网络通信

1.2.3. 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：

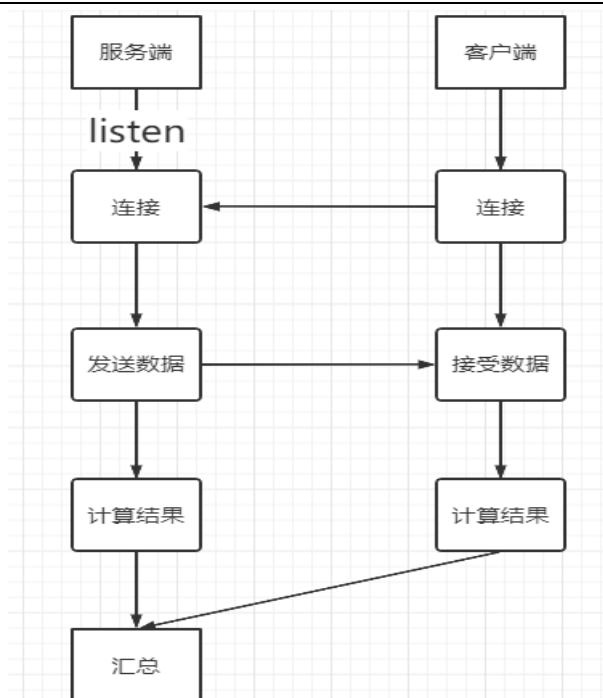


图 1-6 应用的流程图

1.3. SSE 指令集

1.3.1. 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：

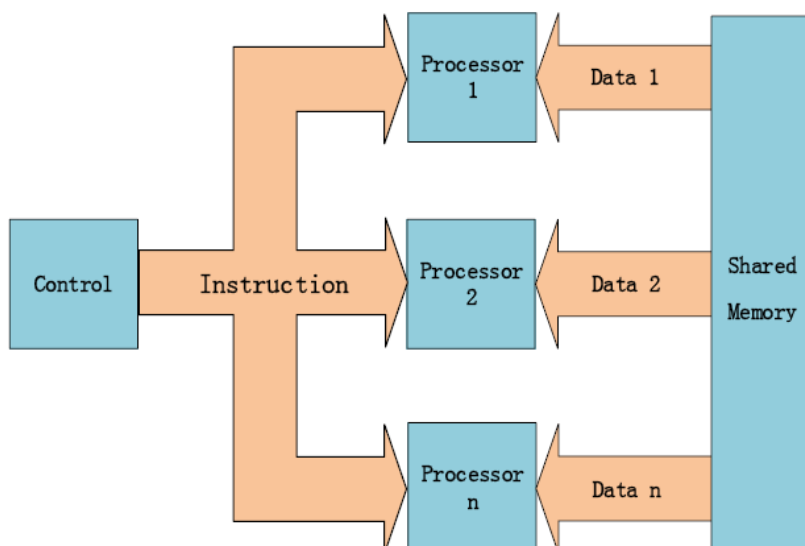


图 1-7 SSE 体系结构

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：

XMM0
XMM1
XMM2
XMM3
XMM4
...

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：

64bit	64bit
-------	-------

或者一次处理 4 个 32bit 的数据类型：

32bit	32bit	32bit	32bit
-------	-------	-------	-------

1.3.2. 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：

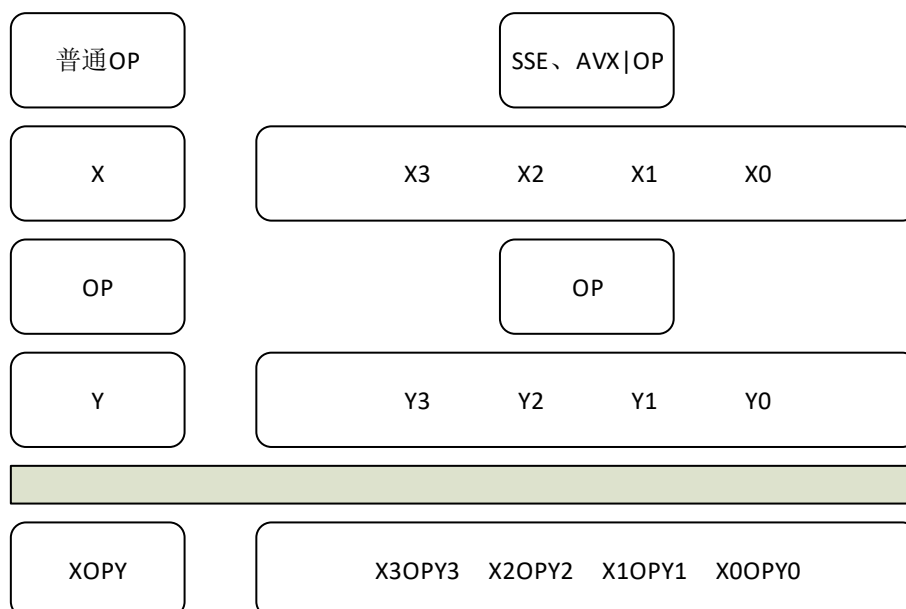


图 1-9 运算对比

- 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。

2. 架构设计

本次大作业的单机的软件架构设计如下图所示。

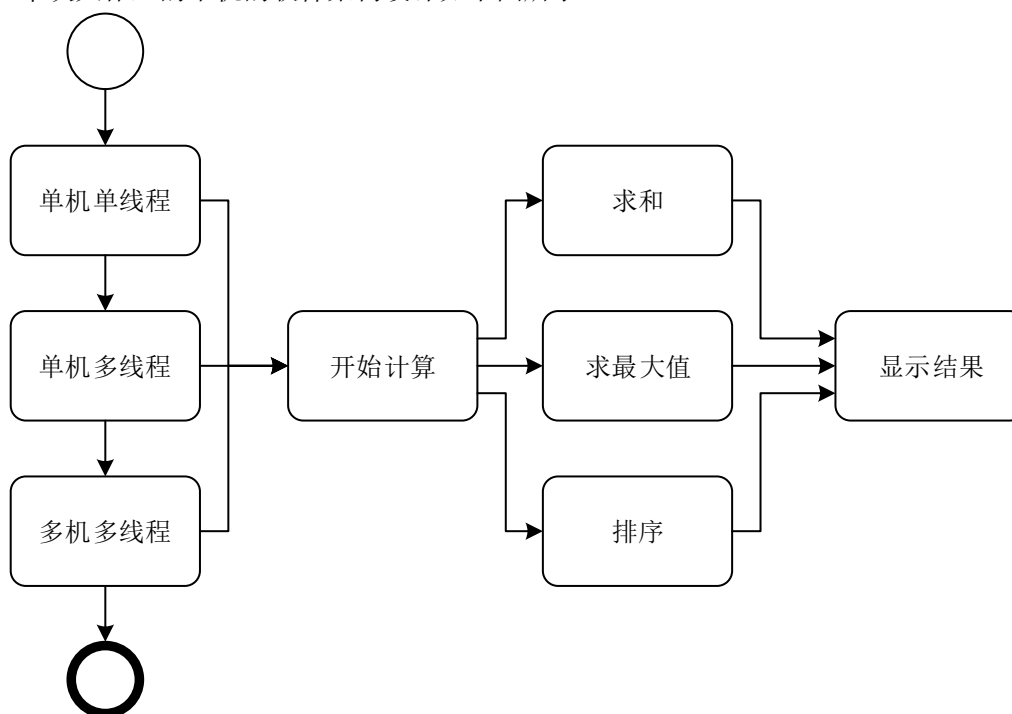


图 2-1 单机软件架构设计

可以从图中看出，单机不论是客户端还是服务端，都依次进行了单机多线程、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。

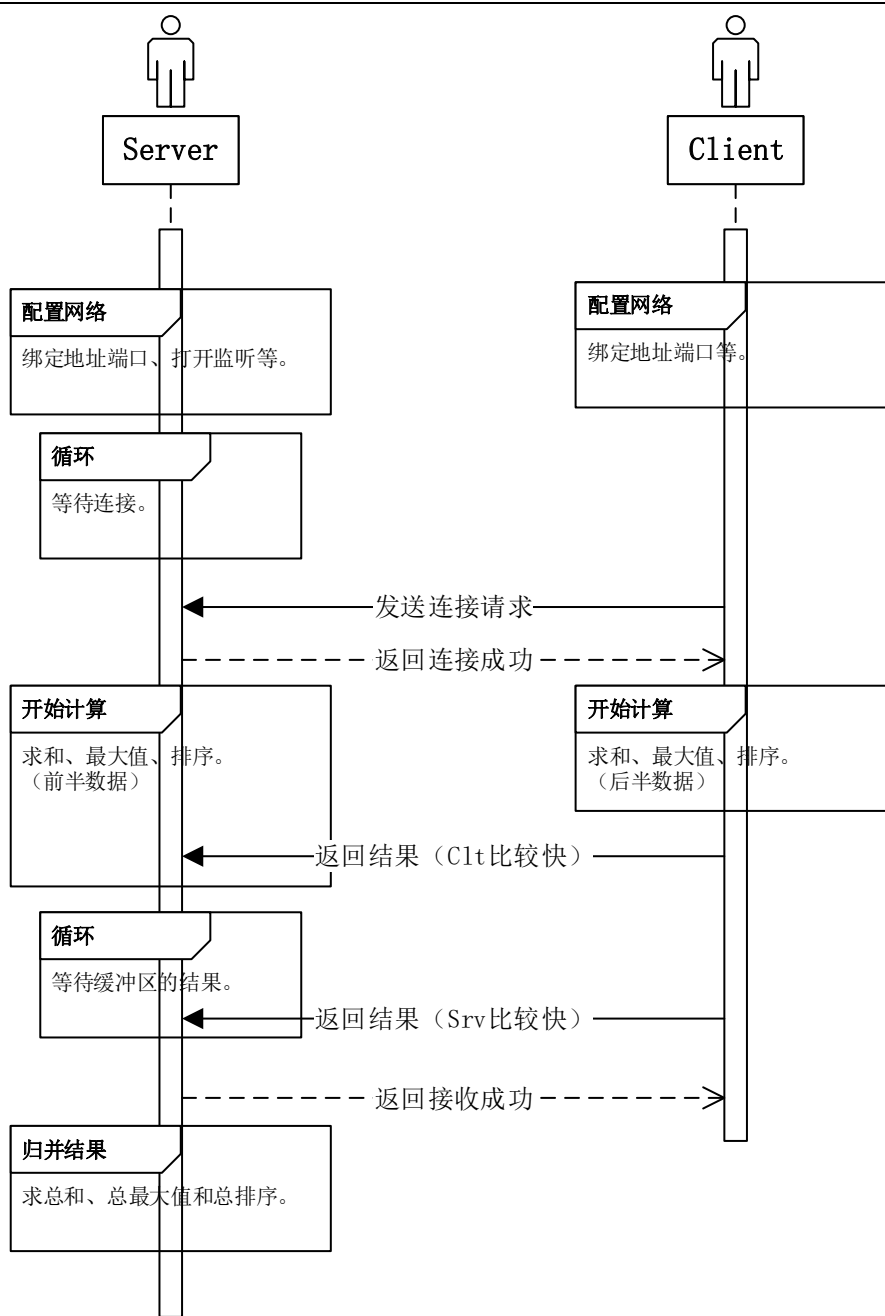


图 2-2 多机流程架构

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。

3. 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。

3.1. 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“mySrvClh.h”，里面声明定义了两端均会使用的变量、函数。

3.1.1. 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
4.
5. #ifndef CLIENT
6. #define SERVER_IP  "127.0.0.1" // 服务端 IP 地址
7. #endif
8.
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。

3.1.2. 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
7.
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
6.
7. #ifdef SERVER
8. #define S_SRV_DATANUM      (S_SRV_SUBDATANUM*MAX_THREADS) // SRV 数据量
9. #endif
10.
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
14.
15. #define DATA_MAX           2147483647 // 随机数的最大值
16.
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
22.
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 ($2^{31} - 1$)。（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”。

3.1.3. 求和

3.1.3.1. 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 \log_{10} 和 $\sqrt{\quad}$ ）。

```
1. // 不加速版本求和
2. double NewSum(const double data[], const int len)
3. {
4.     double rlt = 0.0f;
5.
6.     for (int i = 0; i < len; i++) rlt += log10(sqrt(data[i]/4.0));
7.
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。

3.1.3.2. 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;
5.
6.     int nBlockWidth = 2;
7.     int cntBlock = len/nBlockWidth;
8.     int cntRem = len%nBlockWidth;
9.
10.    const double* p = pbuf;
11.    const double* q;
12.
13.    __m128d xfsload;
14.    __m128d xfssum = _mm_setzero_pd();
15.
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.    }
22.
23.    q = (const double*)&xfssum;
24.    for (int i = 0; i < nBlockWidth ; i++)
25.    {
26.        sum += q[i];
27.    }
28.
29.    // 提取剩余的
30.    for (int i = 0; i < cntRem; i++)
31.    {
32.        sum += p[i];
33.    }
34.
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。

3.1.4. 求最大值

3.1.4.1. 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));
5.
6.     for (int i = 1; i < len; i++) if (log10(sqrt(data[i]/4.0)) > max) max =
       log10(sqrt(data[i]/4.0));
7.
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。

3.1.4.2. 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];
5.
6.     for (int i = 1; i < len; i++) if (data[i] > max) max = data[i];
7.
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。

3.1.5. 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。



3.1.5.1. 冒泡法排序

```
1. double NewSort_0(const double data[], const int len, double result[])
2. {
3.     double tmp;
4.
5.     for (int i = 0; i < len; i++) result[i] = data[i];
6.
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 $O(N^2)$ ，输入参数为待排序的数据和长度、结果存放的数组。

3.1.5.2. 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。

3.1.5.3. 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;
6.
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.    }
17.
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。

3.2. 单机多线程

单机多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_u
   sec);
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。

3.3. 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;
4.
5. // 多线程相关
6. pthread_t tid[MAX_THREADS];
7. pthread_attr_t attr;
8. size_t stacksize;
9.
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]);
4.
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
5.
6.     // 索引
7.     int startIndex = who*SUBDATANUM;
8.     int endIndex = startIndex + SUBDATANUM;
9.
10.    for (int i = startIndex, j = 0; i < endIndex; i++, j++) data[j] = rawDou
        bleData[i];
11.
12.    // 等待发令
13.    while (!thread_begin) {}
14.
15.    // 存储结果
16.    doubleResults[who] = NewSum(data, SUBDATANUM);
17.
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);
3.
4. // 收割
5. finalSum = 0.0f;
6. for (int i = 0; i < MAX_THREADS; i++) finalSum += doubleResults[i];
7.
8. gettimeofday(&endv, &endz);
9. t_usec = (endv.tv_sec - startv.tv_sec)*1000000 + (endv.tv_usec - startv.tv_u
    sec);
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。

3.4. 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。

3.4.1. 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;
7.
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. }
6.
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. }
12.
13. // 等待连接
```



```
14. listen(server_fd, 5);
15.
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。

3.4.2. 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;
7.
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. }
6.
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");
13.
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。

3.4.3. 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.            }
18.
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($O(N \lg N)$)。



4. 测试与结果分析

4.1. 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。

4.2. 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。

4.2.1. 测试 1

```
gzy@gzy: ~/Desktop/vs_code/final_work
gzy@gzy:~/Desktop/vs_code/final_work$ ./myServer
Single-PC-single-thread[SUM](Server): answer = 525514372.944852, time = 4144424 us
Single-PC-single-thread[MAX](Server): answer = 4.364935, time = 4098861 us
Single-PC-single-thread[SORT](Server): answer = 1, time = 2075595 us
Single-PC-multi-thread[SUM](Server): answer = 525514372.944987, time = 1002043 us
Single-PC-multi-thread[MAX](Server): answer = 4.364935, time = 1028797 us
Single-PC-multi-thread[SORT](Server): answer = 1, time = 2692078 us
# Accept client 192.168.43.113
# Send: Hello Client!
Srv sum = 262760094.863880
Multi-PC-multi-thread[SUM](Server): answer = 525514372.944989, time = 723513 us
Srv max = 4.364935
Multi-PC-multi-thread[MAX](Server): answer = 4.364935, time = 457919 us
Srv sort = 1, time = 1367766
# Send: Come on!
Multi-PC-multi-thread[SORT](Server): answer = 1, time = 8507669 us
gzy@gzy:~/Desktop/vs_code/final_work$
```

```
lpj@lpj: ~/Desktop/vs_code/a
lpj@lpj:~/Desktop/vs_code/a$ ./myClient
Single-PC-single-thread[SUM](Client): answer = 525514372.944852, time = 4794716 us
Single-PC-single-thread[MAX](Client): answer = 4.364935, time = 4952400 us
Single-PC-single-thread[SORT](Client): answer = 1, time = 2468760 us
Single-PC-multi-thread[SUM](Client): answer = 525514372.944987, time = 923427 us
Single-PC-multi-thread[MAX](Client): answer = 4.364935, time = 1048697 us
Single-PC-multi-thread[SORT](Client): answer = 1, time = 3513159 us
Connect to server!
# Received: Hello Client!
Client sum = 262754278.081109
Client sum result send successfully!
Client max = 4.364935
Client max result send successfully!
Client sort = 1
# Received: Come on!
Client sort result send successfully!
lpj@lpj:~/Desktop/vs_code/a$
```



4.2.2. 测试 2

```
gzy@gzy: ~/Desktop/vs_code/final_work
gzy@gzy:~/Desktop/vs_code/final_work$ ./myServer
Single-PC-single-thread[SUM](Server): answer = 525514372.944852, time = 4160179 us
Single-PC-single-thread[MAX](Server): answer = 4.364935, time = 4548037 us
Single-PC-single-thread[SORT](Server): answer = 1, time = 2084514 us
Single-PC-multi-thread[SUM](Server): answer = 525514372.944987, time = 970730 us
Single-PC-multi-thread[MAX](Server): answer = 4.364935, time = 1021948 us
Single-PC-multi-thread[SORT](Server): answer = 1, time = 2706645 us
# Accept client 192.168.43.113
# Send: Hello Client!
Srv sum = 262760094.863880
Multi-PC-multi-thread[SUM](Server): answer = 525514372.944989, time = 481029 us
Srv max = 4.364935
Multi-PC-multi-thread[MAX](Server): answer = 4.364935, time = 488166 us
Srv sort = 1, time = 1375783
# Send: Come on!
Multi-PC-multi-thread[SORT](Server): answer = 1, time = 8138407 us
gzy@gzy:~/Desktop/vs_code/final_work$
```

```
lpj@lpj: ~/Desktop/vs_code/a
lpj@lpj:~/Desktop/vs_code/a$ ./myClient
Single-PC-single-thread[SUM](Client): answer = 525514372.944852, time = 4786987 us
Single-PC-single-thread[MAX](Client): answer = 4.364935, time = 4786883 us
Single-PC-single-thread[SORT](Client): answer = 1, time = 2413259 us
Single-PC-multi-thread[SUM](Client): answer = 525514372.944987, time = 908086 us
Single-PC-multi-thread[MAX](Client): answer = 4.364935, time = 1216540 us
Single-PC-multi-thread[SORT](Client): answer = 1, time = 3413515 us
Connect to server!
# Received: Hello Client!
Client sum = 262754278.081109
Client sum result send successfully!
Client max = 4.364935
Client max result send successfully!
Client sort = 1
# Received: Come on!
Client sort result send successfully!
lpj@lpj:~/Desktop/vs_code/a$
```



4.2.3. 测试 3

```
gzy@gzy: ~/Desktop/vs_code/final_work
gzy@gzy:~/Desktop/vs_code/final_work$ ./myServer
Single-PC-single-thread[SUM](Server): answer = 525514372.944852, time = 4283598 us
Single-PC-single-thread[MAX](Server): answer = 4.364935, time = 4108261 us
Single-PC-single-thread[SORT](Server): answer = 1, time = 2077502 us
Single-PC-multi-thread[SUM](Server): answer = 525514372.944987, time = 974999 us
Single-PC-multi-thread[MAX](Server): answer = 4.364935, time = 956940 us
Single-PC-multi-thread[SORT](Server): answer = 1, time = 2691612 us
# Accept client 192.168.43.113
# Send: Hello Client!
Srv sum = 262760094.863880
Multi-PC-multi-thread[SUM](Server): answer = 525514372.944989, time = 470046 us
Srv max = 4.364935
Multi-PC-multi-thread[MAX](Server): answer = 4.364935, time = 463655 us
Srv sort = 1, time = 1365273
# Send: Come on!
Multi-PC-multi-thread[SORT](Server): answer = 1, time = 8285981 us
gzy@gzy:~/Desktop/vs_code/final_work$

lpj@lpj: ~/Desktop/vs_code/a
lpj@lpj:~/Desktop/vs_code/a$ ./myClient
Single-PC-single-thread[SUM](Client): answer = 525514372.944852, time = 4758802 us
Single-PC-single-thread[MAX](Client): answer = 4.364935, time = 4773345 us
Single-PC-single-thread[SORT](Client): answer = 1, time = 2413159 us
Single-PC-multi-thread[SUM](Client): answer = 525514372.944987, time = 1015266 us
Single-PC-multi-thread[MAX](Client): answer = 4.364935, time = 1159779 us
Single-PC-multi-thread[SORT](Client): answer = 1, time = 3383408 us
Connect to server!
# Received: Hello Client!
Client sum = 262754278.081109
Client sum result send successfully!
Client max = 4.364935
Client max result send successfully!
Client sort = 1
# Received: Come on!
Client sort result send successfully!
lpj@lpj:~/Desktop/vs_code/a$
```



4.2.4. 测试 4

```
gzy@gzy: ~/Desktop/vs_code/final_work
gzy@gzy:~/Desktop/vs_code/final_work$ ./myServer
Single-PC-single-thread[SUM](Server): answer = 525514372.944852, time = 4095011 us
Single-PC-single-thread[MAX](Server): answer = 4.364935, time = 4079509 us
Single-PC-single-thread[SORT](Server): answer = 1, time = 2078482 us
Single-PC-multi-thread[SUM](Server): answer = 525514372.944987, time = 989307 us
Single-PC-multi-thread[MAX](Server): answer = 4.364935, time = 1021684 us
Single-PC-multi-thread[SORT](Server): answer = 1, time = 2706404 us
# Accept client 192.168.43.113
# Send: Hello Client!
Srv sum = 262760094.863880
Multi-PC-multi-thread[SUM](Server): answer = 525514372.944989, time = 477947 us
Srv max = 4.364935
Multi-PC-multi-thread[MAX](Server): answer = 4.364935, time = 565820 us
Srv sort = 1, time = 1362047
# Send: Come on!
Multi-PC-multi-thread[SORT](Server): answer = 1, time = 8095497 us
gzy@gzy:~/Desktop/vs_code/final_work$
```

```
lpj@lpj: ~/Desktop/vs_code/a
lpj@lpj:~/Desktop/vs_code/a$ ./myClient
Single-PC-single-thread[SUM](Client): answer = 525514372.944852, time = 4756509 us
Single-PC-single-thread[MAX](Client): answer = 4.364935, time = 4768612 us
Single-PC-single-thread[SORT](Client): answer = 1, time = 2413315 us
Single-PC-multi-thread[SUM](Client): answer = 525514372.944987, time = 905114 us
Single-PC-multi-thread[MAX](Client): answer = 4.364935, time = 1199760 us
Single-PC-multi-thread[SORT](Client): answer = 1, time = 3470038 us
Connect to server!
# Received: Hello Client!
Client sum = 262754278.081109
Client sum result send successfully!
Client max = 4.364935
Client max result send successfully!
Client sort = 1
# Received: Come on!
Client sort result send successfully!
lpj@lpj:~/Desktop/vs_code/a$
```



4.2.5. 测试 5

```
gzy@gzy: ~/Desktop/vs_code/final_work
gzy@gzy:~/Desktop/vs_code/final_work$ ./myServer
Single-PC-single-thread[SUM](Server): answer = 525514372.944852, time = 4095968 us
Single-PC-single-thread[MAX](Server): answer = 4.364935, time = 4086457 us
Single-PC-single-thread[SORT](Server): answer = 1, time = 2079143 us
Single-PC-multi-thread[SUM](Server): answer = 525514372.944987, time = 976514 us
Single-PC-multi-thread[MAX](Server): answer = 4.364935, time = 1027868 us
Single-PC-multi-thread[SORT](Server): answer = 1, time = 2699124 us
# Accept client 192.168.43.113
# Send: Hello Client!
Srv sum = 262760094.863880
Multi-PC-multi-thread[SUM](Server): answer = 525514372.944989, time = 536267 us
Srv max = 4.364935
Multi-PC-multi-thread[MAX](Server): answer = 4.364935, time = 478226 us
Srv sort = 1, time = 1376652
# Send: Come on!
Multi-PC-multi-thread[SORT](Server): answer = 1, time = 9152275 us
gzy@gzy:~/Desktop/vs_code/final_work$
```

```
lpj@lpj: ~/Desktop/vs_code/a
lpj@lpj:~/Desktop/vs_code/a$ ./myClient
Single-PC-single-thread[SUM](Client): answer = 525514372.944852, time = 4758970 us
Single-PC-single-thread[MAX](Client): answer = 4.364935, time = 4777835 us
Single-PC-single-thread[SORT](Client): answer = 1, time = 2414138 us
Single-PC-multi-thread[SUM](Client): answer = 525514372.944987, time = 963861 us
Single-PC-multi-thread[MAX](Client): answer = 4.364935, time = 1241994 us
Single-PC-multi-thread[SORT](Client): answer = 1, time = 3381857 us
Connect to server!
# Received: Hello Client!
Client sum = 262754278.081109
Client sum result send successfully!
Client max = 4.364935
Client max result send successfully!
Client sort = 1
# Received: Come on!
Client sort result send successfully!
lpj@lpj:~/Desktop/vs_code/a$
```



4.2.6. 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。

4.2.7. 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。

4.2.8. 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。求得排序的单机多线程平均时间为 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，相对于单机多线程而言，单机多线程的加速比为 0.73，多机多线程的加速比为 0.27。

4.3. 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 $O(N \lg N)$ ，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 $O(N^2)$ 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。

4.4. 附加测试与结果分析

实验的结果截图如下。

4.4.1. 测试 1

```

gzy@gzy: ~/Desktop/vs_code/final_work
gzy@gzy:~/Desktop/vs_code/final_work$ ./myServer
Single-PC-single-thread[SUM](Server): answer = 525514372.944852, time = 4054695 us
Single-PC-single-thread[MAX](Server): answer = 4.364935, time = 4079534 us
Single-PC-single-thread[SORT](Server): answer = 1, time = 4642003 us
Single-PC-multi-thread[SUM](Server): answer = 525514372.944987, time = 813096 us
Single-PC-multi-thread[MAX](Server): answer = 4.364935, time = 819643 us
Single-PC-multi-thread[SORT](Server): answer = 1, time = 15758 us
# Accept client 192.168.43.113
# Send: Hello Client!
Srv sum = 262760094.863880
Multi-PC-multi-thread[SUM](Server): answer = 525514372.944989, time = 663546 us
Srv max = 4.364935
Multi-PC-multi-thread[MAX](Server): answer = 4.364935, time = 361081 us
Srv sort = 1, time = 5166
# Send: Come on!
Multi-PC-multi-thread[SORT](Server): answer = 1, time = 79035 us
gzy@gzy:~/Desktop/vs_code/final_work$

```




```
lpj@lpj: ~/Desktop/vs_code/final_work
lpj@lpj:~/Desktop/vs_code/final_work$ ./myClient
Single-PC-single-thread[SUM](Client): answer = 525514372.944852, time = 4736940 us
Single-PC-single-thread[MAX](Client): answer = 4.364935, time = 4771786 us
Single-PC-single-thread[SORT](Client): answer = 1, time = 5419210 us
Single-PC-multi-thread[SUM](Client): answer = 525514372.944987, time = 1150674 us
Single-PC-multi-thread[MAX](Client): answer = 4.364935, time = 1266425 us
Single-PC-multi-thread[SORT](Client): answer = 1, time = 26031 us
Connect to server!
# Received: Hello Client!
Client sum = 262754278.081109
Client sum result send successfully!
Client max = 4.364935
Client max result send successfully!
Client sort = 1
# Received: Come on!
Client sort result send successfully!
lpj@lpj:~/Desktop/vs_code/final_work$
```




4.4.2. 测试 2

```
gzy@gzy: ~/Desktop/vs_code/final_work
gzy@gzy:~/Desktop/vs_code/final_work$ ./myServer
Single-PC-single-thread[SUM](Server): answer = 525514372.944852, time = 4055509 us
Single-PC-single-thread[MAX](Server): answer = 4.364935, time = 4082857 us
Single-PC-single-thread[SORT](Server): answer = 1, time = 4627004 us
Single-PC-multi-thread[SUM](Server): answer = 525514372.944987, time = 822930 us
Single-PC-multi-thread[MAX](Server): answer = 4.364935, time = 828246 us
Single-PC-multi-thread[SORT](Server): answer = 1, time = 15726 us
# Accept client 192.168.43.113
# Send: Hello Client!
Srv sum = 262760094.863880
Multi-PC-multi-thread[SUM](Server): answer = 525514372.944989, time = 642283 us
Srv max = 4.364935
Multi-PC-multi-thread[MAX](Server): answer = 4.364935, time = 392571 us
Srv sort = 1, time = 5153
# Send: Come on!
Multi-PC-multi-thread[SORT](Server): answer = 1, time = 75880 us
gzy@gzy:~/Desktop/vs_code/final_work$

lpj@lpj: ~/Desktop/vs_code/final_work
lpj@lpj:~/Desktop/vs_code/final_work$ ./myClient
Single-PC-single-thread[SUM](Client): answer = 525514372.944852, time = 4749993 us
Single-PC-single-thread[MAX](Client): answer = 4.364935, time = 4769185 us
Single-PC-single-thread[SORT](Client): answer = 1, time = 5412082 us
Single-PC-multi-thread[SUM](Client): answer = 525514372.944987, time = 1150063 us
Single-PC-multi-thread[MAX](Client): answer = 4.364935, time = 1260018 us
Single-PC-multi-thread[SORT](Client): answer = 1, time = 25784 us
Connect to server!
# Received: Hello Client!
Client sum = 262754278.081109
Client sum result send successfully!
Client max = 4.364935
Client max result send successfully!
Client sort = 1
# Received: Come on!
Client sort result send successfully!
lpj@lpj:~/Desktop/vs_code/final_work$
```



4.4.3. 测试 3

```
gzy@gzy: ~/Desktop/vs_code/final_work
gzy@gzy:~/Desktop/vs_code/final_work$ ./myServer
Single-PC-single-thread[SUM](Server): answer = 525514372.944852, time = 4054495 us
Single-PC-single-thread[MAX](Server): answer = 4.364935, time = 4079476 us
Single-PC-single-thread[SORT](Server): answer = 1, time = 4657451 us
Single-PC-multi-thread[SUM](Server): answer = 525514372.944987, time = 815725 us
Single-PC-multi-thread[MAX](Server): answer = 4.364935, time = 826393 us
Single-PC-multi-thread[SORT](Server): answer = 1, time = 15762 us
# Accept client 192.168.43.113
# Send: Hello Client!
Srv sum = 262760094.863880
Multi-PC-multi-thread[SUM](Server): answer = 525514372.944989, time = 691180 us
Srv max = 4.364935
Multi-PC-multi-thread[MAX](Server): answer = 4.364935, time = 390399 us
Srv sort = 1, time = 6423
# Send: Come on!
Multi-PC-multi-thread[SORT](Server): answer = 1, time = 83713 us
gzy@gzy:~/Desktop/vs_code/final_work$

lpj@lpj: ~/Desktop/vs_code/final_work
lpj@lpj:~/Desktop/vs_code/final_work$ ./myClient
Single-PC-single-thread[SUM](Client): answer = 525514372.944852, time = 4751314 us
Single-PC-single-thread[MAX](Client): answer = 4.364935, time = 4777347 us
Single-PC-single-thread[SORT](Client): answer = 1, time = 5442326 us
Single-PC-multi-thread[SUM](Client): answer = 525514372.944987, time = 1159418 us
Single-PC-multi-thread[MAX](Client): answer = 4.364935, time = 1263954 us
Single-PC-multi-thread[SORT](Client): answer = 1, time = 25542 us
Connect to server!
# Received: Hello Client!
Client sum = 262754278.081109
Client sum result send successfully!
Client max = 4.364935
Client max result send successfully!
Client sort = 1
# Received: Come on!
Client sort result send successfully!
lpj@lpj:~/Desktop/vs_code/final_work$
```



4.4.4. 测试 4

```
gzy@gzy: ~/Desktop/vs_code/final_work
gzy@gzy:~/Desktop/vs_code/final_work$ ./myServer
Single-PC-single-thread[SUM](Server): answer = 525514372.944852, time = 4050867 us
Single-PC-single-thread[MAX](Server): answer = 4.364935, time = 4075396 us
Single-PC-single-thread[SORT](Server): answer = 1, time = 4623778 us
Single-PC-multi-thread[SUM](Server): answer = 525514372.944987, time = 817987 us
Single-PC-multi-thread[MAX](Server): answer = 4.364935, time = 820689 us
Single-PC-multi-thread[SORT](Server): answer = 1, time = 16438 us
# Accept client 192.168.43.113
# Send: Hello Client!
Srv sum = 262760094.863880
Multi-PC-multi-thread[SUM](Server): answer = 525514372.944989, time = 586704 us
Srv max = 4.364935
Multi-PC-multi-thread[MAX](Server): answer = 4.364935, time = 384971 us
Srv sort = 1, time = 4934
# Send: Come on!
Multi-PC-multi-thread[SORT](Server): answer = 1, time = 90425 us
gzy@gzy:~/Desktop/vs_code/final_work$

lpj@lpj: ~/Desktop/vs_code/final_work
lpj@lpj:~/Desktop/vs_code/final_work$ ./myClient
Single-PC-single-thread[SUM](Client): answer = 525514372.944852, time = 4759350 us
Single-PC-single-thread[MAX](Client): answer = 4.364935, time = 4795666 us
Single-PC-single-thread[SORT](Client): answer = 1, time = 5435617 us
Single-PC-multi-thread[SUM](Client): answer = 525514372.944987, time = 1135606 us
Single-PC-multi-thread[MAX](Client): answer = 4.364935, time = 1240946 us
Single-PC-multi-thread[SORT](Client): answer = 1, time = 27367 us
Connect to server!
# Received: Hello Client!
Client sum = 262754278.081109
Client sum result send successfully!
Client max = 4.364935
Client max result send successfully!
Client sort = 1
# Received: Come on!
Client sort result send successfully!
lpj@lpj:~/Desktop/vs_code/final_work$
```



4.4.5. 测试 5

```
gzy@gzy: ~/Desktop/vs_code/final_work
gzy@gzy:~/Desktop/vs_code/final_work$ ./myServer
Single-PC-single-thread[SUM](Server): answer = 525514372.944852, time = 4055300 us
Single-PC-single-thread[MAX](Server): answer = 4.364935, time = 4075620 us
Single-PC-single-thread[SORT](Server): answer = 1, time = 4688123 us
Single-PC-multi-thread[SUM](Server): answer = 525514372.944987, time = 791575 us
Single-PC-multi-thread[MAX](Server): answer = 4.364935, time = 821855 us
Single-PC-multi-thread[SORT](Server): answer = 1, time = 15544 us
# Accept client 192.168.43.113
# Send: Hello Client!
Srv sum = 262760094.863880
Multi-PC-multi-thread[SUM](Server): answer = 525514372.944989, time = 761120 us
Srv max = 4.364935
Multi-PC-multi-thread[MAX](Server): answer = 4.364935, time = 372228 us
Srv sort = 1, time = 4871
# Send: Come on!
Multi-PC-multi-thread[SORT](Server): answer = 1, time = 84592 us
gzy@gzy:~/Desktop/vs_code/final_work$

lpj@lpj: ~/Desktop/vs_code/final_work
lpj@lpj:~/Desktop/vs_code/final_work$ ./myClient
Single-PC-single-thread[SUM](Client): answer = 525514372.944852, time = 4752841 us
Single-PC-single-thread[MAX](Client): answer = 4.364935, time = 4780244 us
Single-PC-single-thread[SORT](Client): answer = 1, time = 5452385 us
Single-PC-multi-thread[SUM](Client): answer = 525514372.944987, time = 1129838 us
Single-PC-multi-thread[MAX](Client): answer = 4.364935, time = 1241363 us
Single-PC-multi-thread[SORT](Client): answer = 1, time = 26509 us
Connect to server!
# Received: Hello Client!
Client sum = 262754278.081109
Client sum result send successfully!
Client max = 4.364935
Client max result send successfully!
Client sort = 1
# Received: Come on!
Client sort result send successfully!
lpj@lpj:~/Desktop/vs_code/final_work$
```

4.4.6. 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， $N_1 = 12800$ ，需要遍历 $N_1^2 = 1.64e+8$ 次；每个线程 $N_2 = 200$ ，需要遍历 $N_2^2 = 4e+4$ ，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 $64 \times 4 = 256$ 。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， $64 \times 100 \times 8 = 50kBytes$ ，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。

4.4.7. SSE 补充

因为项目临近完成时，仍未找到办法使用 \log_{10} 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。



```
gzy@gzy: ~/Desktop/vs_code/final_work
gzy@gzy:~/Desktop/vs_code/final_work$ ./myServer f
Single-PC-single-thread[SUM](Server): answer = 51343016.000000, time = 456014 us
Single-PC-single-thread[MAX](Server): answer = 4.364935, time = 463762 us
Single-PC-single-thread[SUM](Server): answer = 1, time = 2049879 us
Single-PC-multi-thread[SUM](Server): answer = 129877368.000000, time = 65334 us
Single-PC-multi-thread[MAX](Server): answer = 4.364935, time = 79855 us
Single-PC-multi-thread[SUM](Server): answer = 1, time = 2450201 us
# Accept client 127.0.0.1
# Send: Hello Client!
Srv sum = 64939784.000000
Multi-PC-multi-thread[SUM](Server): answer = 129877136.000000, time = 61743 us
Srv max = 4.364935
Multi-PC-multi-thread[MAX](Server): answer = 4.364935, time = 82759 us
Srv sort = 1, time = 1366664
# Send: Come on!
Multi-PC-multi-thread[SUM](Server): answer = 1, time = 1462769 us

gzy@gzy:~/Desktop/vs_code/final_work
gzy@gzy:~/Desktop/vs_code/final_work$ ./myClient f
Single-PC-single-thread[SUM](Client): answer = 51343016.000000, time = 449263 us
Single-PC-single-thread[MAX](Client): answer = 4.364935, time = 438910 us
Single-PC-single-thread[SUM](Client): answer = 1, time = 2091769 us
Single-PC-multi-thread[SUM](Client): answer = 129877368.000000, time = 60455 us
Single-PC-multi-thread[MAX](Client): answer = 4.364935, time = 79909 us
Single-PC-multi-thread[SUM](Client): answer = 1, time = 2472300 us
Connect to server!
# Received: Hello Client!
Client sum = 64937348.000000
Client sum result send successfully!
Client max = 4.364935
Client max result send successfully!
Client sort = 1
# Received: Come on!
Client sort result send successfully!
```

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。

5. 总结

5.1. 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。

5.2. 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！



6. 重现时注意

6.1. 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。

6.2. IP 地址更改

6.2.1. IP 地址存放所属文件

IP 地址存放于头文件 mySrvClh.h 中，如图所示：

mySrvClh	2021/1/8 15:46	C/C++ Header	6 KB
----------	----------------	--------------	------

6.2.2. IP 地址修改位置

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

```
23 #ifndef CLIENT  
24 #define SERVER_IP "172.20.10.7" // 服务端IP地址  
25 #endif
```

6.2.3. 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。

6.3. 运算数据量更改

6.3.1. 运算数据量存放位置

运算数据量存放于头文件 mySrvClh.h 中，如图所示：

mySrvClh.h 2021/1/8 15:46 C/C++ Header 6 KB

6.3.2. 运算功能数据量修改位置

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

```

27 // 线程相关
28 #define MAX_THREADS 64 // 线程数: 64
29 #define SUBDATANUM 2000000 // 子块数据量: 2000000
30 #define SRV_SUBDATANUM 1000000 // 单PC数据
31 #define CLT_SUBDATANUM 1000000 // 单PC数据
32 #define DATANUM (SUBDATANUM*MAX_THREADS) // 总数据量: 线程数x子块数据量
  
```

6.3.3. 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)

6.4. 排序功能数据量更改

6.4.1. 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 mySrvClh.h 中，如图所示：

mySrvClh.h 2021/1/8 15:46 C/C++ Header 6 KB

6.4.2. 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 mySrvClh.h 头文件的第 34 行至第 42 行的位置的位置，如图所示：

```

34 #define S_SUBDATANUM 200 // 减少数据量进行排序
35 #define S_SRV_SUBDATANUM 100 // 单PC小数据量测试
36 #define S_CLT_SUBDATANUM 100 // 单PC小数据量测试
37 #define S_DATANUM (S_SUBDATANUM*MAX_THREADS) // 总数据量(小): 线程数x子块数据量
38 #define S_CLT_DATANUM (S_CLT_SUBDATANUM*MAX_THREADS) // CLT数据量
39
40 #ifdef SERVER
41 #define S_SRV_DATANUM (S_SRV_SUBDATANUM*MAX_THREADS) // SRV数据量
42 #endif
  
```

6.4.3. 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子块数据量
S_CLT_SUBDATANUM	100	[1, 100000]	排序时客户端每个线程的子块数据量
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。

6.5. socket 传输数据量更改

6.5.1. socket 传输数据量存放位置

socket 传输数据量存放于头文件 mySrvClh.h 中，如图所示：

 mySrvClh.h	2021/1/8 15:46	C/C++ Header	6 KB
--	----------------	--------------	------

6.5.2. socket 传输数据量修改位置

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

```

44 #define S_ONCE          100          // 一次发送100个double
45 #define S_TIMES        (S_CLT_DATANUM/S_ONCE) // 总共发送100个的次数
46 #define S_LEFT          (S_CLT_DATANUM%S_ONCE) // 发送剩余不足的数据

```

6.5.3. 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)

6.5.4. 排序算法存放位置

排序算法存放于头文件 mySrvClh.h 中，如图所示：

 mySrvClh.h	2021/1/8 15:46	C/C++ Header	6 KB
--	----------------	--------------	------

6.5.5. 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：



```
200 void* fnThreadSort(void* arg)
201 {
202     int who = *(int *)arg;
203     double data[S_SRV_SUBDATANUM];
204
205     // 索引
206     int startIndex = who*S_SRV_SUBDATANUM;
207     int endIndex = startIndex + S_SRV_SUBDATANUM;
208
209     for (int i = startIndex, j = 0; i < endIndex; i++, j++) data[j] = S_rawDoubleData[i];
210
211     while (!thread_begin) {}
212
213     NewSort_0(data, S_SRV_SUBDATANUM, S_threadResult[who]);
214 }
```

6.5.6. 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：

```
182 double NewSort_0(const double data[], const int len, double result[])
183 {
184     double tmp;
185
186     for (int i = 0; i < len; i++) result[i] = data[i];
187
188     for (int i = 0; i < len; i++)
189     {
190         for (int j = 0; j < len - i - 1; j++)
191         {
192             if (log10(sqrt(result[j])) > log10(sqrt(result[j + 1])))
193             {
194                 tmp = result[j];
195                 result[j] = result[j + 1];
196                 result[j+1] = tmp;
197             }
198         }
199     }
200 }
201
```

- (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：

```
171 // 不加速排序（快排）
172 double NewSort(const double data[], const int len, double result[])
173 {
174     int l = 0, r = len - 1;
175     for (int i = 0; i < len; i++)
176     {
177         result[i] = data[i];
178     }
179     qsort(result, l, r);
180 }
```

6.5.7. 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：

```

167 void* fnThreadSort(void* arg)
168 {
169     int who = *(int *)arg;
170     double data[S_CLT_SUBDATANUM];
171
172     // 索引
173     int startIndex = who*S_CLT_SUBDATANUM + S_SRV_SUBDATANUM*MAX_THREADS;
174     int endIndex = startIndex + S_CLT_SUBDATANUM;
175
176     for (int i = startIndex, j = 0; i < endIndex; i++, j++) data[j] = S_rawDoubleData[i];
177
178     while (!thread_begin) {}
179
180     NewSort_0(data, S_CLT_SUBDATANUM, S_threadResult[who]);
181 }

```

6.5.8. 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

7. 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

8. 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[J] Computers and Electrical Engineering, 2018, 65.