

# 同濟大學

TONGJI UNIVERSITY

## 多机-多线程-SSE 运算加速

课程名称 系统编程/并行编程

学生姓名 郭展羽 学 号 2230730

学生姓名 王渤 学 号 2232970

任课教师 王晓年

日 期 2022 年 12 月 29 日



## 目 录

1. 加速原理.....	1
1.1. 多线程 .....	1
1.2. 多机通信 .....	3
1.3. SSE 指令集 .....	5
2. 架构设计.....	8
3. 代码实现.....	10
3.1. 公用部分 .....	10
3.2. 单机单线程 .....	16
3.3. 单机多线程 .....	16
3.4. 多机多线程 .....	18
3.5. 单机 Python 语言实现.....	22
3.6. Python 中调用 C++程序.....	24
4. 实验.....	27
4.1. 测试环境 .....	27
4.2. 测试结果 .....	27
4.3. 结果分析 .....	32
5. 总结.....	34
6. 重现时的注意事项.....	35
6.1. 编译 .....	35
6.2. IP 地址 .....	35
6.3. 运算数据量 .....	35
6.4. 排序数据量 .....	36
6.5. 传输数据量 .....	36
7. 分工.....	37
参考文献.....	38



## 1. 加速原理

本次大作业使用到的加速方法有：多线程、多机通信和 SSE 指令集，下文将依次将上述原理进行介绍和分析。

### 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：

名称	状态	14% CPU	39% 内存	50% 磁盘	0% 网络
<strong>应用 (4)</strong>					
Microsoft Word (32 位) (2)		0.2%	154.2 MB	0 MB/秒	0 Mbps
WeChat (32 位) (10)		2.1%	299.8 MB	0 MB/秒	0 Mbps
Windows 资源管理器		0.8%	62.4 MB	0 MB/秒	0 Mbps
任务管理器		1.0%	30.9 MB	0 MB/秒	0 Mbps
<strong>后台进程 (128)</strong>					
64-bit Synaptics Pointing Enhanc...		0%	1.7 MB	0 MB/秒	0 Mbps
AcroTray (32 位)		0%	1.1 MB	0 MB/秒	0 Mbps
Activation Licensing Service		0%	1.3 MB	0 MB/秒	0 Mbps
Adobe Acrobat Update Service (...)		0%	0.6 MB	0 MB/秒	0 Mbps
Adobe Genuine Software Integrit...		0%	1.0 MB	0 MB/秒	0 Mbps
Antimalware Service Executable		3.5%	229.5 MB	0.1 MB/秒	0 Mbps
AppHelperCap.exe		0%	5.3 MB	0 MB/秒	0 Mbps

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。比如此时我同时用腾讯 QQ 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 1.2 所示。

腾讯QQ (32 位) (2)	5.2%	151.5 MB	0.4 MB/秒	3.3 Mbps
控制系2022级硕士教学群				
图片查看				

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。如下图所示，主线程可以和许多工作线程并发执行。

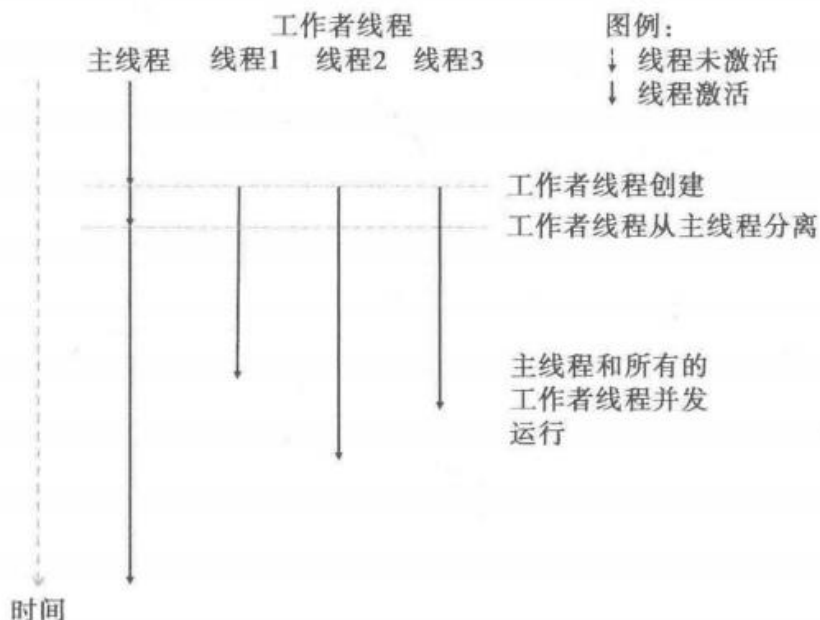


图 1.3 多线程例子

### 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 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，由于 CPU 在多个线程间切换速度较快，看上去就是在同一时刻运行。

实际上，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。主线程与其他线程的关系如下图 1.4 所示。

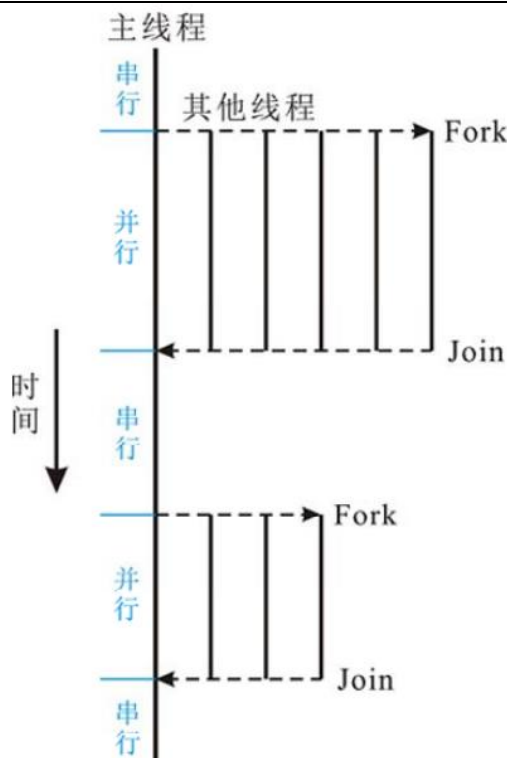


图 1.4 主线程与其他线程的关系

在多线程程序中，当主线程必须等待时，CPU 可以运行其它的工作线程，而不是等待，从而极大提升了程序的运行效率。这便是多线程的加速原理。

## 1.2. 多机通信

### 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，并行计算框架可划分为先对任务进行划分，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 实际为一个门面模式，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，有时应用需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。所谓流是指不间断的数据结构，当应用程序采用 TCP 发送消息时，其即可以保证发送的顺序，且可以将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”的一种机制。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。综上所述，借助于 Socket 的网络通信如下图 1.5 所示。

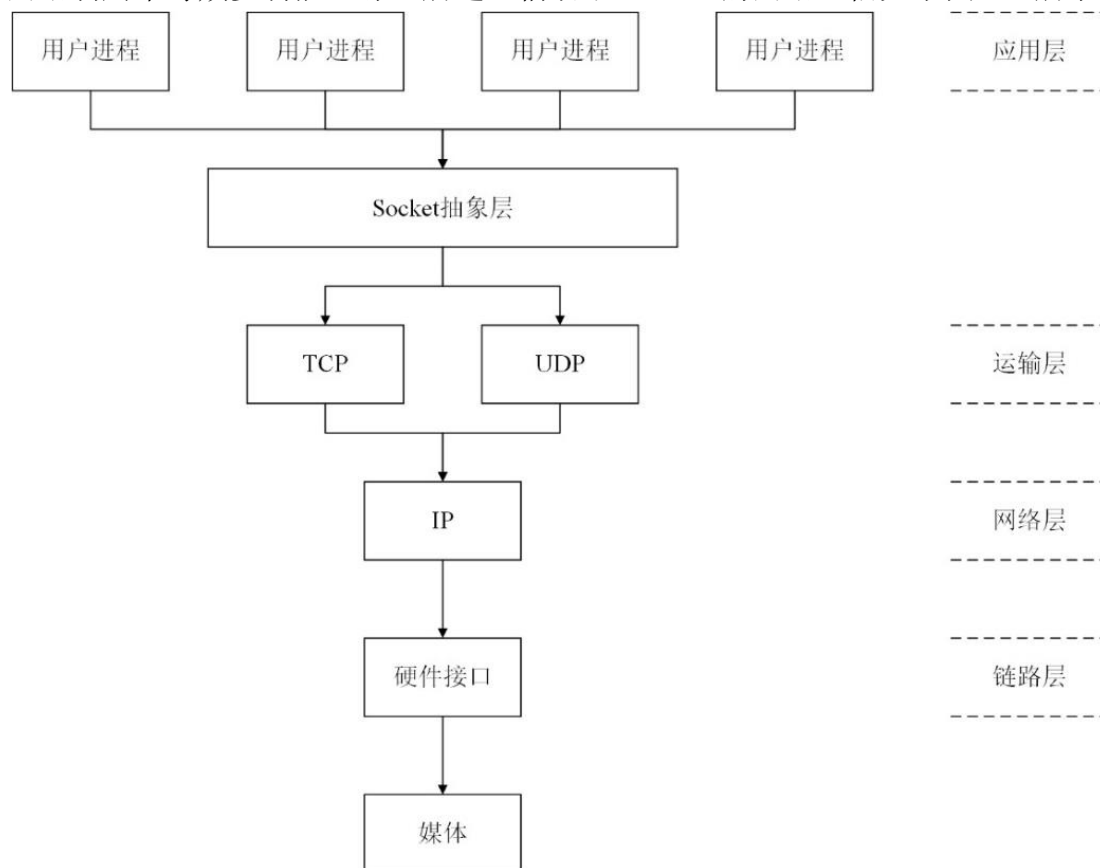


图 1.5 Socket 网络通信

### 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。其具体流程图如图 1.6 所示。

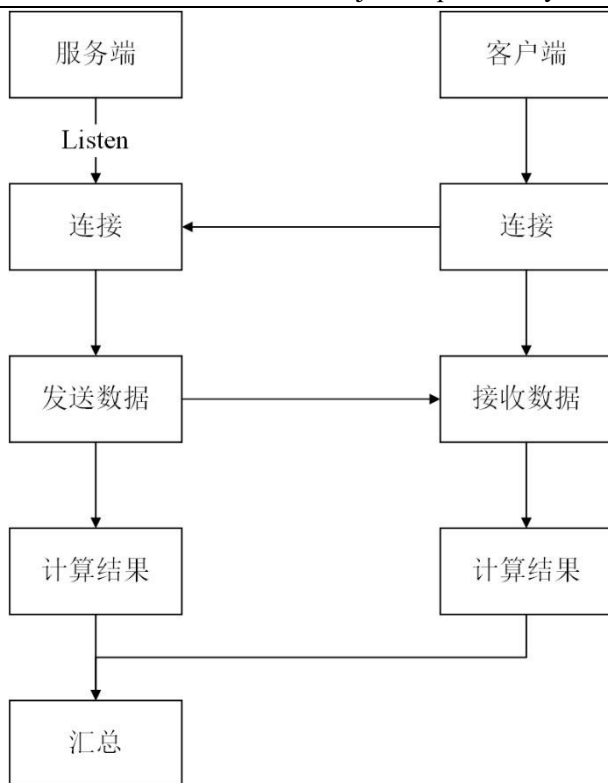


图 1.6 多机并行计算应用流程

### 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 1.7 所示。

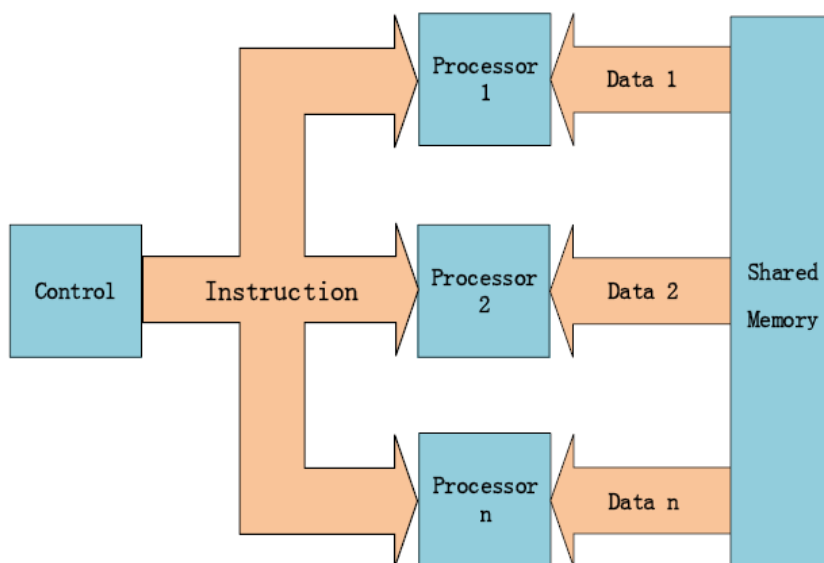


图 1.7 SSE 体系结构



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。寄存器结构如图 1.8 所示。

XMM0
XMM1
XMM2
XMM3
XMM4
...

图 1.8 寄存器结构

由于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 1.9 所示。

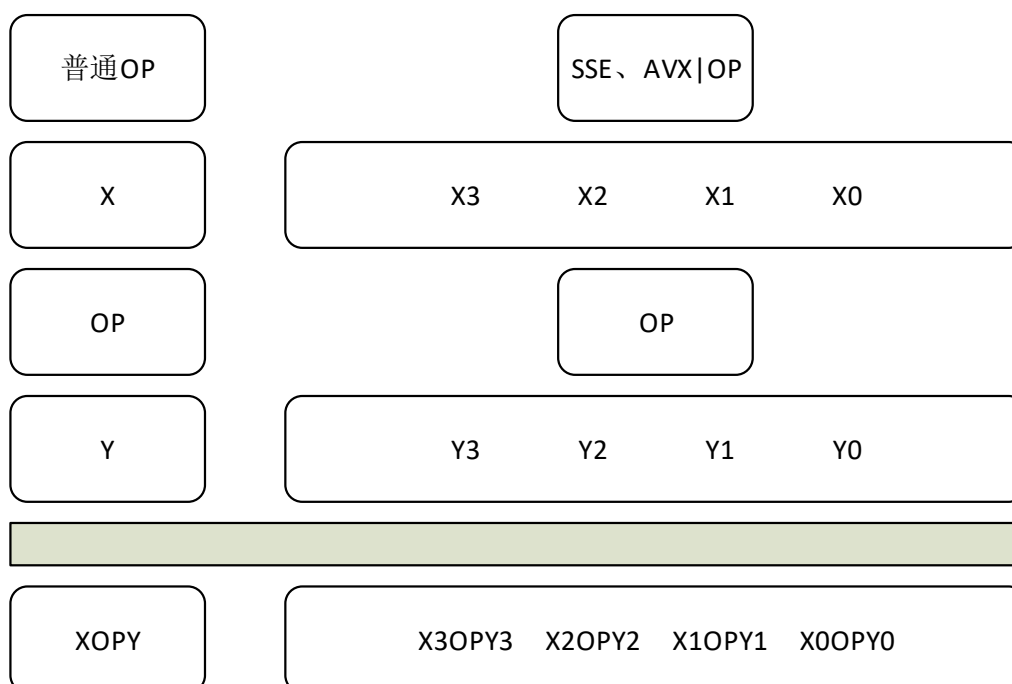


图 1.9 运算对比





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  $X \text{op} Y$ 。
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  $X0 \text{op} Y0$ 、 $X1 \text{op} Y1$ 、 $X2 \text{op} Y2$ 、 $X3 \text{op} Y3$ 。

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。这就是使用 SSE 指令集可以实现算法加速的原理。



## 2. 架构设计

本次课程设计的单机的软件架构设计如图 2.1 所示。

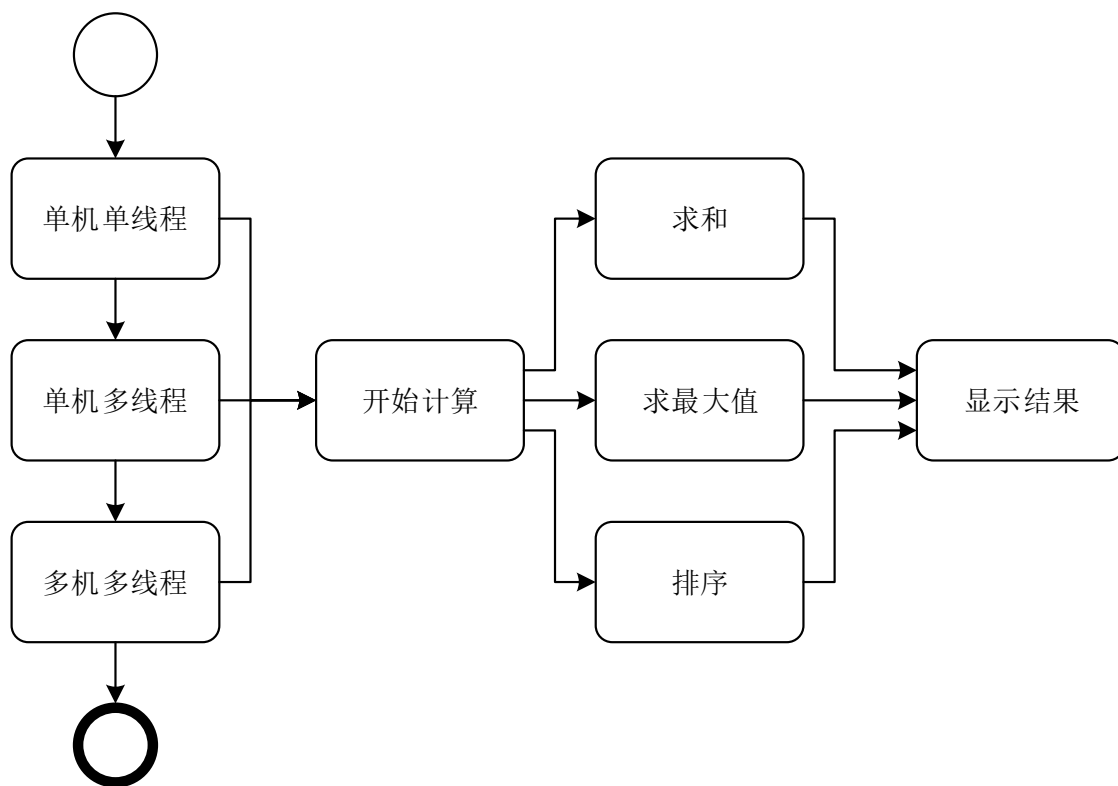


图 2.1 单机软件架构设计

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 2.2 所示。

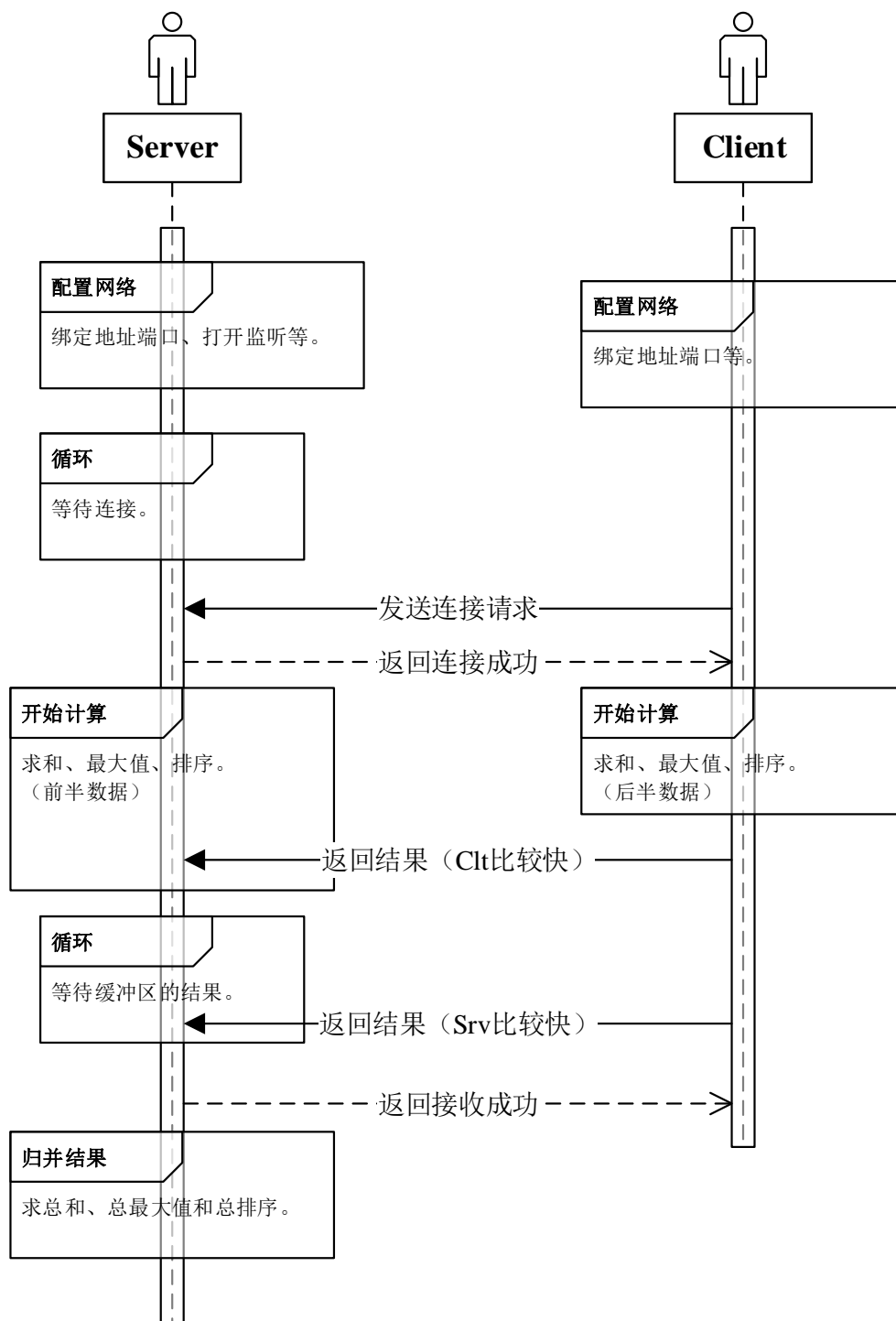


图 2.2 多机流程架构

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

本次大作业在 Linux 系统下实现，具体的版本为 Ubuntu 20.04 LTS。大作业的实现可以分为三部分：单机单线程、单机多线程和多机多线程，然后我们在 Python 中实现部分功能并和直接调用 C++代码进行对比。在对各部分详细介绍之前，有许多宏定义和共用的函数，需要首先进行说明。

#### 3.1. 公用部分

为了方便在调试时进行更改，客户端（Client）和服务端（Server）会共同调用一个头文件“mySrvClnt.hpp”，里面声明定义了两方均会使用到的变量和函数。

##### 3.1.1. 网络通信部分

```
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];
```

网络端口号定义为 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，则还需定义服务端的 IP 地址，用以绑定套接字进行连接。

##### 3.1.2. 多线程部分

```
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
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; // 求和结果
12. double finalMax; // 求最大值最终结果
```

线程数定义为 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。其中总的数量等于线程数和子块数据量的乘积。用以上的宏定义, 创建分别用于存储原始数据、各线程结果(求和与求最大值)、求和最终结果与求最大值最终结果的全局变量。因为排序所需的时间较长, 为了方便调试并在不影响前两者的情况下减小数据量, 所以将排序的各种参数与前面两者分离, 便于数据的管理和代码的调试。具体的定义如下所示。

```
1. #define S_SUBDATANUM 400 // 减少数据量进行排序
2. #define S_SRV_SUBDATANUM 200 // 单 PC 小数据量测试
3. #define S_CLT_SUBDATANUM 200 // 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 1600 // 一次发送 1600 个 double
12. #define S_TIMES (S_CLT_DATANUM / S_ONCE) // 总共发送 1600 个的次数
13. #define S_LEFT (S_CLT_DATANUM % S_ONCE) // 发送剩余不足的数据
14.
15. #define DATA_MAX 2147483647 // 随机数可能产生的最大值
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_threadResult[MAX_THREADS][S_SUBDATANUM]; // 单机多线程各线程结果
20. double S_threadResult0[MAX_THREADS][S_SRV_SUBDATANUM]; // 多机多线程各线程结果
21.
22. double S_CLT_sortDoubleData[S_CLT_DATANUM]; // CLT 排序最终结果
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 ( $2^{31} - 1$ )。(注意: Linux 中, C++ 随机数生成范围不同于 Windows, 范围从 -INT\_MAX+1 到 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。

为了保证多线程运算时间的统计公平, 应该使得多个线程同时开始。然而 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。实际上，经过我们的测试，当线程数目较多时，这种方式非常占用系统资源，但在 64 线程的条件下影响不大。具体的实现方法见以下代码。

标志位（全局变量）：

```
1. // 标志位
2. bool thread_begin; // 线程发令标志
```

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. 基础无加速版本

不加速的求和方法就是简单的循环和累加，为了延长计算的时间，累加的同时对数据进行运算处理（取 `log10` 和 `sqrt()`）。

```
1. // 不加速版本求和
2. double sum(const double data[], const int len)
3. {
4.     double result = 0.0;
5.
6.     for (int i = 0; i < len; i++)
7.         result += log10(sqrt(data[i]));
8.
9.     return result;
10. }
```

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，所以 128 位可以同时两组进行运算，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 sumSSE(const double *pbuf, const int len)
3. {
4.     double sum = 0.0;
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))); // 两个并行计算，使用
        sqrt 代替 log10
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.        sum += q[i];
26.
27.    // 提取剩余的
28.    for (int i = 0; i < cntRem; i++)
29.        sum += p[i];
30.
31.    return sum;
32. }
```

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。

由于未找到办法对双精度浮点型（double）进行 log10 的 SSE 加速运算，于是只能使用一个 sqrt 进行代替。实际测试时为了对照结果的准确性，会将基础求



和部分进行相应的更改。

### 3.1.4. 求最大值

#### 3.1.4.1. 作数据处理的求最大值

求最大值的方法非常简单，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 max(const double data[], const int len)
3. {
4.     double result = log10(sqrt(data[0]));
5.
6.     for (int i = 1; i < len; i++)
7.         if (log10(sqrt(data[i])) > result)
8.             result = log10(sqrt(data[i]));
9.
10.    return result;
11. }
```

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 maxMerge(const double data[], const int len)
3. {
4.     double result = data[0];
5.
6.     for (int i = 1; i < len; i++)
7.         if (data[i] > result)
8.             result = data[i];
9.
10.    return result;
11. }
```

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

本次大作业使用的排序算法为冒泡法排序，相比于快速排序算法，能在多线程、多机具有更明显的加速效果。算法的实现如下所示。





### 3.1.5.1. 冒泡法排序

```
1. // 不加速排序（冒泡）
2. void sort(const double data[], const int len, double result[])
3. {
4.     double tmp;
5.
6.     for (int i = 0; i < len; i++)
7.         result[i] = data[i];
8.
9.     for (int i = 0; i < len; i++)
10.        for (int j = 0; j < len - i - 1; j++)
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. }
```

这是在 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。注意这里为了增加处理时间，在遍历的同时也对数值作类似于求和部分的处理。

### 3.1.5.2. 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所示。

```
1. // 检验结果
2. bool isSorted(const double data[], const int len)
3. {
4.     double sign, new_sign;
5.
6.     sign = data[1] - data[0];
7.     for (int i = 1; i < len - 1; i++)
8.     {
9.         new_sign = data[i + 1] - data[i];
10.        if (new_sign * sign < 0)
11.            return false;
12.    }
13.
14.    return true;
15. }
```



该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，输出 1 为有序，0 为无序。

### 3.2. 单机单线程

单机单线程的运算流程实现十分简单，即直接将生成的数据放入到函数中进行完全遍历，所以耗时也将会是最长。这里以求和为例，其余的运算类似。

```
1. finalSum = 0.0;
2. gettimeofday(&startv, &startz);
3. finalSum = sum(rawDoubleData, DATANUM);
4. gettimeofday(&endv, &endz);
5. t_usec = (endv.tv_sec - startv.tv_sec)*1000000+(endv.tv_usec - startv.tv_usec);
6. printf("Single-PC-single-
   thread[SUM](Server): answer = %lf, time = %ld us\n", finalSum, t_usec);
```

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

### 3.3. 单机多线程

同样地，单机多线程的实现在这里只阐述求和的方法，其余的运算类似。

```
1. // 每个线程的 ID
2. int id[MAX_THREADS];
3.
4. // 给每个线程一个 ID 号地址，保证 ID 号在创建线程时不会发生冲突
5. for (int i = 0; i < MAX_THREADS; i++)
6.     id[i] = i;
7.
8. // 多线程相关
9. pthread_t tid[MAX_THREADS];
10. pthread_attr_t attr;
11. size_t stacksize;
12.
13. // 更改栈的大小
14. pthread_attr_init(&attr);
15. pthread_attr_getstacksize(&attr, &stacksize);
16. stacksize *= 4;
17. pthread_attr_setstacksize(&attr, stacksize);
```

首先需要给每个线程分配 ID，之后会作为参数传进每个线程函数中。然后初始化一些跟多线程相关的变量，用以后续创建线程。在我们实现的过程中，遇



到了线程堆栈大小不足的情况。我们将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 -----
2. thread_begin = false;
3. for (int i = 0; i < MAX_THREADS; i++)
4.     pthread_create(&tid[i], &attr, fnThreadSum, &id[i]);
5.
6. // 给多个线程同时发令
7. gettimeofday(&starttv, &starttz);
8. 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. 在主线程创建线程时，每创建一个线程后挂起一小段时间，保证子线程能读出正确的 ID。缺点为：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(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.    // 数据转移
11.    for (int i = startIndex, j = 0; i < endIndex; i++, j++)
12.        data[j] = rawDoubleData[i];
13.
14.    // 等待发令
15.    while (!thread_begin);
16.
17.    // 存储结果
18.    doubleResults[who] = sum(data, SUBDATANUM);
```



```
19.  
20.     return NULL;  
21. }
```

可以看到，线程函数中获取了 ID 值，并得到本线程对应的数据段，对该段数据求和，存放在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++)  
3.     pthread_join(tid[i], NULL);  
4.  
5. // 收割  
6. finalSum = 0.0;  
7. for (int i = 0; i < MAX_THREADS; i++)  
8.     finalSum += doubleResults[i];  
9.  
10. gettimeofday(&endv, &endz);  
11. t_usec = (endv.tv_sec - startv.tv_sec)*1000000+(endv.tv_usec - startv.tv_usec);  
12. printf("Single-PC-multip-  
    thread[SUM](Server): answer = %lf, time = %ld us\n", finalSum, t_usec);
```

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，与开始时间作差得出计算所耗费的时间。

### 3.4. 多机多线程

在单机多线程的基础上，可以进行多机协作，进一步加快计算的速度。在这一部分，只介绍流程最为复杂的排序。无论是多机还是单机，只要是将数据分段进行排序的运算，到最后都会遇到结果归并的问题。各段虽然有序，但各段之间的关系式无序的，这种排序的结果实际上还不如快速排序的中间态。这也是我们使用冒泡排序才能体现加速效果的原因。

我们先对网络进行了配置和连接，分为服务端（Server）和客户端（Client）两种不同的操作，将在下文详细介绍。

#### 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...\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...\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...\n");
21.     return -1;
22. }
23. printf("# Accept client %s\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...\n");
```



```
7.     return -1;
8. }
9. printf("# Send: Hello Client!\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...\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...\n");
10.    return -1;
11. }
12. printf("Connect to server!\n");
```

绑定套接字，然后就可以尝试连接前面定义好的地址。

```
1. if ((recv_len = recv(client_fd, buf, BUF_LEN, 0)) < 0)
2. {
```



```
3.     printf("client recv failed...\n");
4.     return -1;
5. }
6. printf("# Received: %s\n", buf);
```

连接成功后，等待接收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

随后的流程为双机各自开启多线程运算，在这里不再进行赘述。在本节中我们将介绍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.         index[i] = 0;
6.
7.     int min_index;
8.     for (int j = 0; j < S_DATANUM; j++)
9.     {
10.        double min_num = log10(sqrt(DATA_MAX));
11.        for (int i = 0; i < MAX_THREADS; i++)
12.        {
13.            if (index[i] > S_SUBDATANUM - 1)
14.                continue;
15.
16.            if (S_threadResult[i][index[i]] < min_num)
17.            {
18.                min_index = i;
19.                min_num = S_threadResult[i][index[i]];
20.            }
21.        }
22.        S_sortDoubleData[j] = min_num;
23.        index[min_index]++;
24.    }
25. }
```

假设有 64 线程，在多线程得到 64 组排序结果后，每组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，但也大幅减少了  $O(N^2)$  时间复杂度的冒泡排序算法带来的消耗。

同样的思路，将双机的结果进行归并，可以得到最后的排序结果。但由于需



要将客户端所有的排序结果发送到服务端，数据的传输所耗时间占了不少的比重，同时又对传输的质量有很高的需求。

### 3.5. 单机 Python 语言实现

在单机中使用 Python 语言实现了与 C++ 类似的求最大值运算，使用到了多种方案并进行对比：单线程、多线程、多进程和 NumPy 库。接下来将对这些方案分别进行介绍。

#### 3.5.1. 单线程

```
1. import math
2.
3. def maxNormal(ls):
4.     result = -math.inf
5.     for item in ls:
6.         a = math.log10(math.sqrt(item))
7.         if a > result:
8.             result = a
9.
10.    return result
```

使用 math 库进行 log10 和 sqrt 的运算，类似于 C++ 实现中进行边遍历边对数据进行处理。使用以下函数进行单线程的求最大值运算。

```
1. from time import time
2.
3. # python normal
4. start = time()
5. result = maxNormal(rawDoubleData)
6. end = time()
7. time_nm = end - start
8. print("python single threads: answer = {:.3f}, duration = {:.4f} s".format(result, time_nm))
```

在主函数中使用 time 进行计时并输出结果。

#### 3.5.2. 多线程

```
1. from threading import Thread
2.
3. MAX_THREADS = 64
4. SUBDATANUM = 2000000
5. DATANUM = MAX_THREADS * SUBDATANUM
6.
```





```
7. # python acc 多线程
8. threads = []
9. doubleResults = [0 for _ in range(MAX_THREADS)]
10. for i in range(MAX_THREADS):
11.     t = Thread(target=fnThreadMax, args=(i, ))
12.     t.setDaemon(True)
13.     threads.append(t)
```

依次创建 64 个线程并存入线程列表，每个线程得到线程 ID 值，线程函数如下所示。

```
1. def fnThreadMax(who):
2.     global rawDoubleData, doubleResults
3.
4.     startIndex = who * SUBDATANUM
5.     endIndex = startIndex + SUBDATANUM
6.
7.     result = -math.inf
8.     for i in range(startIndex, endIndex):
9.         a = math.log10(math.sqrt(rawDoubleData[i]))
10.        if a > result:
11.            result = a
12.
13.    doubleResults[who] = result
```

将求得的最大值存入本线程所属的结果中。

```
1. start = time()
2. for t in threads:
3.     t.start()
4. for t in threads:
5.     t.join()
6. result = maxMerge(doubleResults)
7. end = time()
8. time_acc = end - start
9. print("python multip threads: answer = {:.3f}, duration = {:.4f} s".format(result, time_acc))
```

开始计时的同时开启所有的线程，并等待所有线程执行完毕。进行最大值结果归并后输出结果和所消耗的时间以供对比。其中最大值归并的函数如下所示。

```
1. def maxMerge(data):
2.     result = -math.inf
3.     for item in data:
4.         if item > result:
```



```
5.         result = item
6.
7.     return result
```

使用单纯的求最大值求得多个线程最大值的最终结果。

### 3.5.3. 多进程

由于 Python 中的多线程并不能充分地利用多核 CPU（具体实验和分析见“4.3.2. Python 调用对比分析”），我们使用多进程来对该问题进行加速。多进程与多线程的运行流程类似，不过需要额外构造一个共享变量来存储最终结果。

```
1. from multiprocessing import Process, Array
2.
3. # python acc 多进程
4. threads = []
5. doubleResults = Array('d', [0 for _ in range(MAX_THREADS)])
6. for i in range(MAX_THREADS):
7.     t = Process(target=fnThreadMax, args=(i, ))
8.     t.daemon = True
9.     threads.append(t)
```

其余部分与多线程相同。

### 3.5.4. Numpy

```
1. import numpy as np
2.
3. # numpy
4. arr = np.array(rawDoubleData)
```

首先将数据转换为 NumPy 中的数组类型。

```
1. start = time()
2. result = np.amax(np.log10(np.sqrt(arr)))
3. end = time()
4. time_np = end - start
5. print("numpy: answer = {:.3f}, duration = {:.4f} s".format(result, time_np))
```

计时并进行相应的数据处理。输出结果和所消耗的时间以作对比。

## 3.6. Python 中调用 C++ 程序

在 Python 中可以使用 Ctypes 对 C++ 程序编译的动态链接库进行调用，首先需要使用 C++ 编写测试函数。



### 3.6.1. C++测试函数的编写

为了使外部能够调用 C++编写的函数,首先需要使得所有编写的代码包含在 `extern "C"`内, 例如:

```
1. extern "C"
2. {
3.     // coding here
4. }
```

首先我们进行数据的初始化,同时供外部调用获取我们初始化的数据,以保持内外数据的一致性。

```
1. double *initData()
2. {
3.     for (int i = 0; i < DATANUM; i++)
4.         rawDoubleData[i] = fabs(rand());
5.
6.     return rawDoubleData;
7. }
```

然后就是编写单线程的求最大值函数,直接调用公用头文件的函数即可。

```
1. double maxNormal()
2. {
3.     return max(rawDoubleData, DATANUM);
4. }
```

多线程函数也是同理,按照上文的流程编写即可,供外部程序调用的函数名为 `maxAcc`。

### 3.6.2. 动态链接库的编译

直接使用 `g++`对该 `cpp` 文件进行编译,即

```
$ g++ -o max_test.so -shared -fPIC cpp2python.cpp
```

得到一个名为“`max_test.so`”的动态链接库供 Python 调用。

### 3.6.3. Python 中的调用和测试

```
1. import os
2. import ctypes
3.
4. # load library
5. lib_name = "max_test"
6. lib_path = os.path.join(os.path.dirname(__file__), "{}/{}".format(lib_name, ".so"))
7. lib = ctypes.cdll.LoadLibrary(os.path.abspath(lib_path))
```

首先使用 `Ctypes` 读取刚刚编译好的动态链接库。



```
1. # init data, from cpp to py
2. lib.initData.restype = ctypes.POINTER(ctypes.c_double)
3. data = lib.initData()
4. rawDoubleData = [data[i] for i in range(DATANUM)]
```

从 C++ 程序中获得初始化数据的 double 型指针，将其转化为 Python 中的列表供 Python 中其它方法使用。

```
1. # cpp normal
2. lib.maxNormal.restype = ctypes.c_double
3. start = time()
4. result = lib.maxNormal()
5. end = time()
6. time_cpp_normal = end - start
7. print("cpp normal: answer = {:.3f}, duration = {:.2f} s".format(result, time_cpp_normal))
```

调用动态链接库中的 maxNormal 函数测试单线程结果，对 C++ 中的多线程调用也是类似，将函数名更改为 maxAcc 即可。



## 4. 实验

### 4.1. 测试环境

操作系统：Server 和 Client 均为 Linux 系统，具体为 Ubuntu 20.04 LTS；

处理器：Server 为 Intel(R) Core(TM) i7-12700H CPU 2.3GHz，14 核；

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

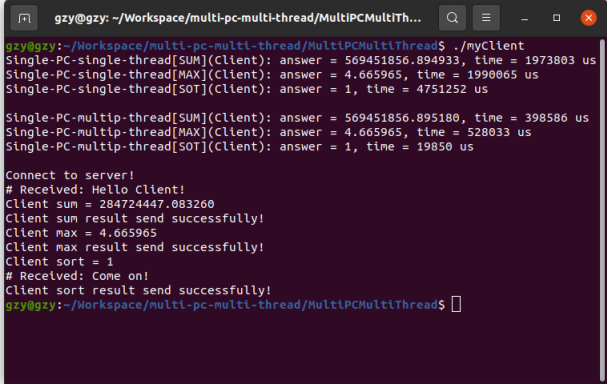
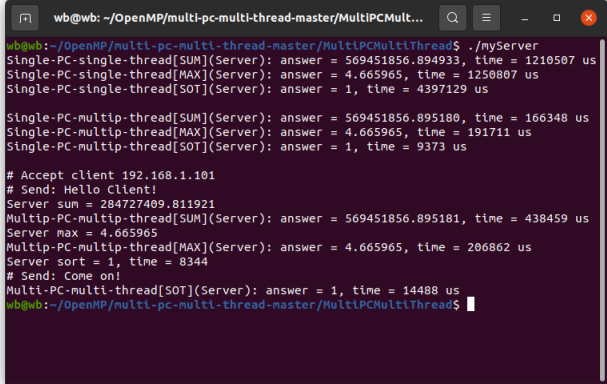
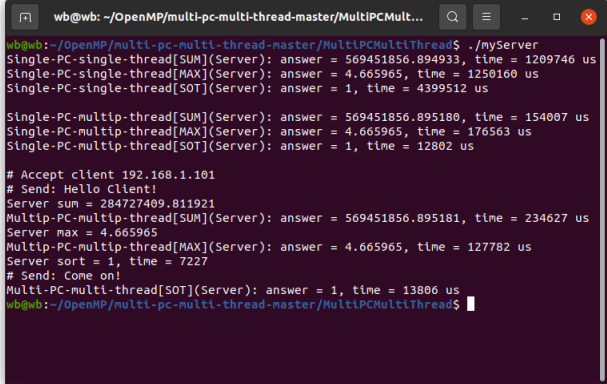
编译器：gcc 5.4.0。

### 4.2. 测试结果

本次课程设计报告中，测试实验结果以图片和表格数据两种形式展示分析。

#### 4.2.1. 多机多线程测试结果

表 4.1 多机多线程图片测试结果

序号	客户端	服务端
1		
2		



# 系统编程/并行编程课程项目报告

## Project Report for System/Parallel Programming

3

```
gz@gz: ~/Workspace/multi-pc-multi-thread/MultiPCMultiTh...
gz@gz:~/Workspace/multi-pc-multi-thread/MultiPCMultiThread$ ./myClient
Single-PC-single-thread[SUM](Client): answer = 569451856.894933, time = 1956558 us
Single-PC-single-thread[MAX](Client): answer = 4.665965, time = 1980951 us
Single-PC-single-thread[SOT](Client): answer = 1, time = 4690285 us

Single-PC-multip-thread[SUM](Client): answer = 569451856.895180, time = 394623 us
Single-PC-multip-thread[MAX](Client): answer = 4.665965, time = 492966 us
Single-PC-multip-thread[SOT](Client): answer = 1, time = 21898 us

Connect to server!
# Received: Hello Client!
Client sum = 284724447.083260
Client sum result send successfully!
Client max = 4.665965
Client max result send successfully!
Client sort = 1
# Received: Come on!
Client sort result send successfully!
gz@gz:~/Workspace/multi-pc-multi-thread/MultiPCMultiThread$
```

```
wb@wb: ~/OpenMP/multi-pc-multi-thread-master/MultiPCMultiTh...
wb@wb:~/OpenMP/multi-pc-multi-thread-master/MultiPCMultiThread$ ./myServer
Single-PC-single-thread[SUM](Server): answer = 569451856.894933, time = 1210556 us
Single-PC-single-thread[MAX](Server): answer = 4.665965, time = 1258565 us
Single-PC-single-thread[SOT](Server): answer = 1, time = 4406058 us

Single-PC-multip-thread[SUM](Server): answer = 569451856.895180, time = 158476 us
Single-PC-multip-thread[MAX](Server): answer = 4.665965, time = 175783 us
Single-PC-multip-thread[SOT](Server): answer = 1, time = 10928 us

# Accept client 192.168.1.101
# Send: Hello Client!
Server sum = 284727409.811921
Multi-PC-multip-thread[SUM](Server): answer = 569451856.895181, time = 374163 us
Server max = 4.665965
Multi-PC-multip-thread[MAX](Server): answer = 4.665965, time = 129314 us
Server sort = 1, time = 8794
# Send: Come on!
Multi-PC-multip-thread[SOT](Server): answer = 1, time = 14751 us
wb@wb:~/OpenMP/multi-pc-multi-thread-master/MultiPCMultiThread$
```

4

```
gz@gz: ~/Workspace/multi-pc-multi-thread/MultiPCMultiTh...
gz@gz:~/Workspace/multi-pc-multi-thread/MultiPCMultiThread$ ./myClient
Single-PC-single-thread[SUM](Client): answer = 569451856.894933, time = 1948243 us
Single-PC-single-thread[MAX](Client): answer = 4.665965, time = 1993442 us
Single-PC-single-thread[SOT](Client): answer = 1, time = 4599588 us

Single-PC-multip-thread[SUM](Client): answer = 569451856.895180, time = 388659 us
Single-PC-multip-thread[MAX](Client): answer = 4.665965, time = 470303 us
Single-PC-multip-thread[SOT](Client): answer = 1, time = 19364 us

Connect to server!
# Received: Hello Client!
Client sum = 284724447.083260
Client sum result send successfully!
Client max = 4.665965
Client max result send successfully!
Client sort = 1
# Received: Come on!
Client sort result send successfully!
gz@gz:~/Workspace/multi-pc-multi-thread/MultiPCMultiThread$
```

```
wb@wb: ~/OpenMP/multi-pc-multi-thread-master/MultiPCMultiTh...
wb@wb:~/OpenMP/multi-pc-multi-thread-master/MultiPCMultiThread$ ./myServer
Single-PC-single-thread[SUM](Server): answer = 569451856.894933, time = 1208687 us
Single-PC-single-thread[MAX](Server): answer = 4.665965, time = 1258705 us
Single-PC-single-thread[SOT](Server): answer = 1, time = 4404802 us

Single-PC-multip-thread[SUM](Server): answer = 569451856.895180, time = 159373 us
Single-PC-multip-thread[MAX](Server): answer = 4.665965, time = 201197 us
Single-PC-multip-thread[SOT](Server): answer = 1, time = 9543 us

# Accept client 192.168.1.101
# Send: Hello Client!
Server sum = 284727409.811921
Multi-PC-multip-thread[SUM](Server): answer = 569451856.895181, time = 313212 us
Server max = 4.665965
Multi-PC-multip-thread[MAX](Server): answer = 4.665965, time = 291034 us
Server sort = 1, time = 8683
# Send: Come on!
Multi-PC-multip-thread[SOT](Server): answer = 1, time = 14211 us
wb@wb:~/OpenMP/multi-pc-multi-thread-master/MultiPCMultiThread$
```

5

```
gz@gz: ~/Workspace/multi-pc-multi-thread/MultiPCMultiTh...
gz@gz:~/Workspace/multi-pc-multi-thread/MultiPCMultiThread$ ./myClient
Single-PC-single-thread[SUM](Client): answer = 569451856.894933, time = 1997706 us
Single-PC-single-thread[MAX](Client): answer = 4.665965, time = 1992320 us
Single-PC-single-thread[SOT](Client): answer = 1, time = 4640199 us

Single-PC-multip-thread[SUM](Client): answer = 569451856.895180, time = 478123 us
Single-PC-multip-thread[MAX](Client): answer = 4.665965, time = 426464 us
Single-PC-multip-thread[SOT](Client): answer = 1, time = 19457 us

Connect to server!
# Received: Hello Client!
Client sum = 284724447.083260
Client sum result send successfully!
Client max = 4.665965
Client max result send successfully!
Client sort = 1
# Received: Come on!
Client sort result send successfully!
gz@gz:~/Workspace/multi-pc-multi-thread/MultiPCMultiThread$
```

```
wb@wb: ~/OpenMP/multi-pc-multi-thread-master/MultiPCMultiTh...
wb@wb:~/OpenMP/multi-pc-multi-thread-master/MultiPCMultiThread$ ./myServer
Single-PC-single-thread[SUM](Server): answer = 569451856.894933, time = 1209598 us
Single-PC-single-thread[MAX](Server): answer = 4.665965, time = 1253682 us
Single-PC-single-thread[SOT](Server): answer = 1, time = 4407399 us

Single-PC-multip-thread[SUM](Server): answer = 569451856.895180, time = 142471 us
Single-PC-multip-thread[MAX](Server): answer = 4.665965, time = 196959 us
Single-PC-multip-thread[SOT](Server): answer = 1, time = 10550 us

# Accept client 192.168.1.101
# Send: Hello Client!
Server sum = 284727409.811921
Multi-PC-multip-thread[SUM](Server): answer = 569451856.895181, time = 294882 us
Server max = 4.665965
Multi-PC-multip-thread[MAX](Server): answer = 4.665965, time = 340300 us
Server sort = 1, time = 8130
# Send: Come on!
Multi-PC-multip-thread[SOT](Server): answer = 1, time = 13955 us
wb@wb:~/OpenMP/multi-pc-multi-thread-master/MultiPCMultiThread$
```

6

```
gz@gz: ~/Workspace/multi-pc-multi-thread/MultiPCMultiTh...
gz@gz:~/Workspace/multi-pc-multi-thread/MultiPCMultiThread$ ./myClient
Single-PC-single-thread[SUM](Client): answer = 569451856.894933, time = 1968479 us
Single-PC-single-thread[MAX](Client): answer = 4.665965, time = 1990592 us
Single-PC-single-thread[SOT](Client): answer = 1, time = 4698005 us

Single-PC-multip-thread[SUM](Client): answer = 569451856.895180, time = 469363 us
Single-PC-multip-thread[MAX](Client): answer = 4.665965, time = 497157 us
Single-PC-multip-thread[SOT](Client): answer = 1, time = 19593 us

Connect to server!
# Received: Hello Client!
Client sum = 284724447.083260
Client sum result send successfully!
Client max = 4.665965
Client max result send successfully!
Client sort = 1
# Received: Come on!
Client sort result send successfully!
gz@gz:~/Workspace/multi-pc-multi-thread/MultiPCMultiThread$
```

```
wb@wb: ~/OpenMP/multi-pc-multi-thread-master/MultiPCMultiTh...
wb@wb:~/OpenMP/multi-pc-multi-thread-master/MultiPCMultiThread$ ./myServer
Single-PC-single-thread[SUM](Server): answer = 569451856.894933, time = 1210165 us
Single-PC-single-thread[MAX](Server): answer = 4.665965, time = 1254325 us
Single-PC-single-thread[SOT](Server): answer = 1, time = 4401794 us

Single-PC-multip-thread[SUM](Server): answer = 569451856.895180, time = 145384 us
Single-PC-multip-thread[MAX](Server): answer = 4.665965, time = 187875 us
Single-PC-multip-thread[SOT](Server): answer = 1, time = 8850 us

# Accept client 192.168.1.101
# Send: Hello client!
Server sum = 284727409.811921
Multi-PC-multip-thread[SUM](Server): answer = 569451856.895181, time = 284615 us
Server max = 4.665965
Multi-PC-multip-thread[MAX](Server): answer = 4.665965, time = 136108 us
Server sort = 1, time = 8608
# Send: Come on!
Multi-PC-multip-thread[SOT](Server): answer = 1, time = 14553 us
wb@wb:~/OpenMP/multi-pc-multi-thread-master/MultiPCMultiThread$
```



7

```
gz@gz: ~/Workspace/multi-pc-multi-thread/MultiPCMultiTh...
gz@gz:~/Workspace/multi-pc-multi-thread/MultiPCMultiThread$ ./myClient
Single-PC-single-thread[SUM](Client): answer = 569451856.894933, time = 1971512 us
Single-PC-single-thread[MAX](Client): answer = 4.665965, time = 2804831 us
Single-PC-single-thread[SOT](Client): answer = 1, time = 4690398 us

Single-PC-multip-thread[SUM](Client): answer = 569451856.895180, time = 422364 us
Single-PC-multip-thread[MAX](Client): answer = 4.665965, time = 490098 us
Single-PC-multip-thread[SOT](Client): answer = 1, time = 28448 us

Connect to server!
# Received: Hello Client!
Client sum = 284724447.083260
Client sum result send successfully!
Client max = 4.665965
Client max result send successfully!
Client sort = 1
# Received: Come on!
Client sort result send successfully!
gz@gz:~/Workspace/multi-pc-multi-thread/MultiPCMultiThread$
```

```
wb@wb: ~/OpenMP/multi-pc-multi-thread-master/MultiPCMultiTh...
wb@wb:~/OpenMP/multi-pc-multi-thread-master/MultiPCMultiThread$ ./myServer
Single-PC-single-thread[SUM](Server): answer = 569451856.894933, time = 1211454 us
Single-PC-single-thread[MAX](Server): answer = 4.665965, time = 1252358 us
Single-PC-single-thread[SOT](Server): answer = 1, time = 4403804 us

Single-PC-multip-thread[SUM](Server): answer = 569451856.895180, time = 132635 us
Single-PC-multip-thread[MAX](Server): answer = 4.665965, time = 197202 us
Single-PC-multip-thread[SOT](Server): answer = 1, time = 10031 us

# Accept client 192.168.1.101
# Send: Hello Client!
Server sum = 284727409.811921
Multi-PC-multip-thread[SUM](Server): answer = 569451856.895181, time = 328449 us
Server max = 4.665965
Multi-PC-multip-thread[MAX](Server): answer = 4.665965, time = 244780 us
Server sort = 1, time = 7901
# Send: Come on!
Multi-PC-multip-thread[SOT](Server): answer = 1, time = 16292 us
wb@wb:~/OpenMP/multi-pc-multi-thread-master/MultiPCMultiThread$
```

表 4.2 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	1973803	1210507	1592155	398586	166348	282467	438459
2	1959859	1209746	1584803	393192	154007	273600	234627
3	1956558	1210556	1583557	394623	158476	276500	374163
4	1948243	1208687	1578465	388659	159373	274016	313212
5	1997706	1209598	1603652	478123	142471	310297	294882
6	1968479	1210165	1589322	469363	145384	307375	284615
7	1971512	1211454	1591483	422364	132635	277500	328449
平均			1589545			288558	304991
加速比			1.0000			5.5086	5.2118

以上时间单位为 us，由于第 1、3 组数据偏离于其他数据，故将其剔除。所求得求和 SUM 的单机单线程平均时间为 1589545us，单机多线程平均时间为 288558us，多机多线程平均时间为 304991us。得出作求和 SUM 运算时，相对于单机单线程而言，单机多线程的加速比为 5.5086，多机多线程的加速比为 5.2118。

表 4.3 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	1990065	1250807	1620436	528033	191711	359872	206862
2	1982797	1250160	1616479	481799	176563	329181	127782
3	1980951	1250565	1615758	492966	175783	334375	129314
4	1993442	1250705	1622074	470303	201197	335750	291034
5	1992320	1253682	1623001	426464	196959	311712	340300
6	1990592	1254325	1622459	497157	187875	342516	136108
7	2004831	1252358	1628595	490098	197202	343650	244780
平均			1620745			341919	168969
加速比			1.0000			4.7401	9.5920





以上时间单位为 us，由于第 4、5 组数据偏离于其他数据，故将其剔除。所求得最大值 MAX 的单机多线程平均时间为 1620745us，单机多线程平均时间为 341919us，多机多线程平均时间为 168969us。得出作求最大值 MAX 运算时，相对于单机多线程而言，单机多线程的加速比为 4.7401，多机多线程的加速比为 9.5920。

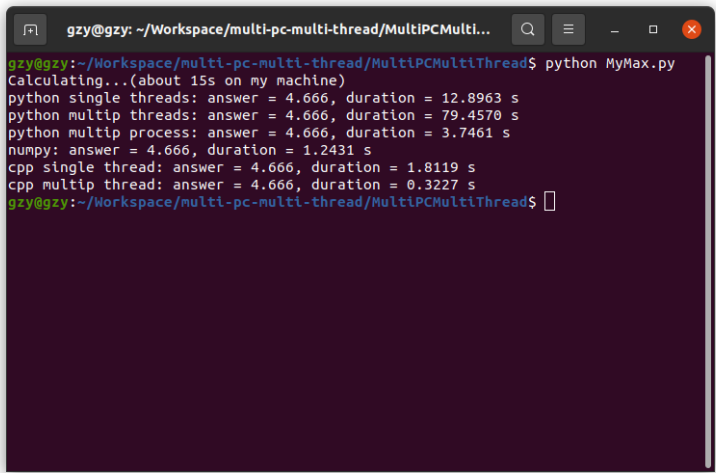
表 4.4 排序结果

序号	A 单机多线程 SORT	B 单机多线程 SORT	平均单机多线程 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	4751252	4397129	4574191	19850	9323	14587	14488
2	5625238	4399512	5012375	29952	12802	21377	13806
3	4690285	4406058	4548172	21898	10928	16413	14751
4	4599588	4404002	4501795	19364	9543	14454	14221
5	4640199	4407399	4523799	19457	10550	15004	13955
6	4698005	4401794	4549900	19593	8850	14222	14553
7	4690398	4403804	4547101	20448	10031	15240	16292
平均			4539571			14936	14394
加速比			1.0000			303.93	315.38

以上时间单位为 us，由于第 2、7 组数据偏离于其他数据，故将其剔除。所求得排序 SORT 的单机多线程平均时间为 4539571us，单机多线程平均时间为 14936us，多机多线程平均时间为 14394us。得出作排序 SORT 运算时，相对于单机多线程而言，单机多线程的加速比为 303.93，多机多线程的加速比为 315.38。

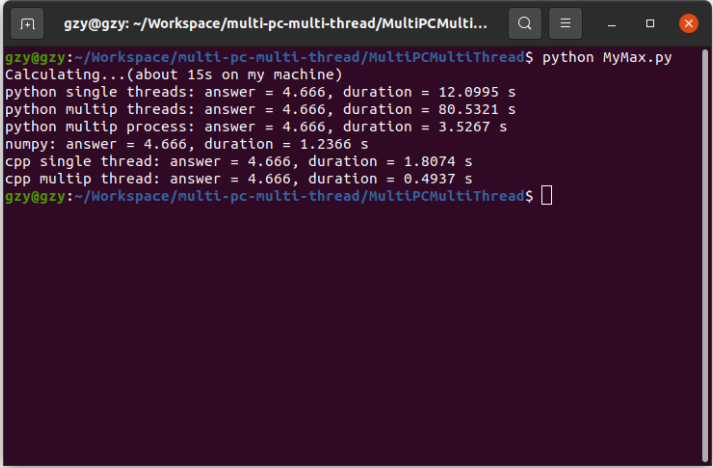
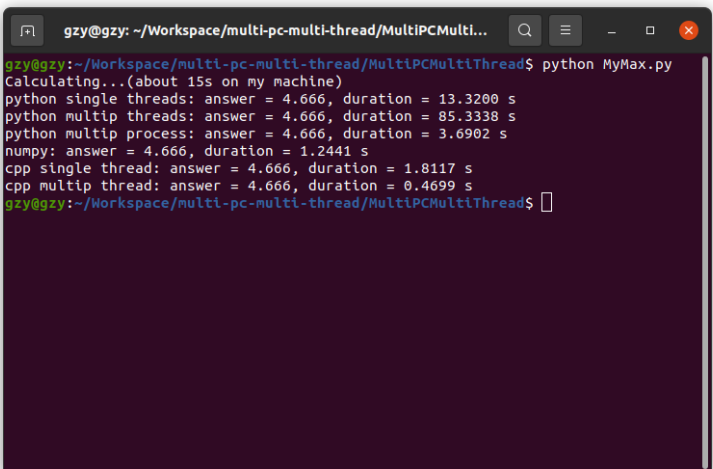
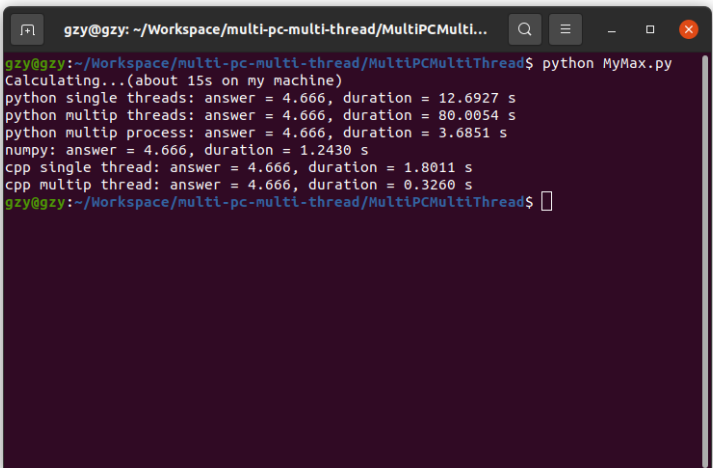
#### 4.2.2. Python 实现结果

表 4.5 Python 图片测试结果

序号	结果
1	





2	 <pre>gzy@gzy: ~/Workspace/multi-pc-multi-thread/MultiPCMultiThread\$ python MyMax.py Calculating...(about 15s on my machine) python single threads: answer = 4.666, duration = 12.0995 s python multip threads: answer = 4.666, duration = 80.5321 s python multip process: answer = 4.666, duration = 3.5267 s numpy: answer = 4.666, duration = 1.2366 s cpp single thread: answer = 4.666, duration = 1.8074 s cpp multip thread: answer = 4.666, duration = 0.4937 s gzy@gzy:~/Workspace/multi-pc-multi-thread/MultiPCMultiThread\$</pre>
3	 <pre>gzy@gzy:~/Workspace/multi-pc-multi-thread/MultiPCMultiThread\$ python MyMax.py Calculating...(about 15s on my machine) python single threads: answer = 4.666, duration = 13.3200 s python multip threads: answer = 4.666, duration = 85.3338 s python multip process: answer = 4.666, duration = 3.6902 s numpy: answer = 4.666, duration = 1.2441 s cpp single thread: answer = 4.666, duration = 1.8117 s cpp multip thread: answer = 4.666, duration = 0.4699 s gzy@gzy:~/Workspace/multi-pc-multi-thread/MultiPCMultiThread\$</pre>
4	 <pre>gzy@gzy:~/Workspace/multi-pc-multi-thread/MultiPCMultiThread\$ python MyMax.py Calculating...(about 15s on my machine) python single threads: answer = 4.666, duration = 12.6927 s python multip threads: answer = 4.666, duration = 80.0054 s python multip process: answer = 4.666, duration = 3.6851 s numpy: answer = 4.666, duration = 1.2430 s cpp single thread: answer = 4.666, duration = 1.8011 s cpp multip thread: answer = 4.666, duration = 0.3260 s gzy@gzy:~/Workspace/multi-pc-multi-thread/MultiPCMultiThread\$</pre>



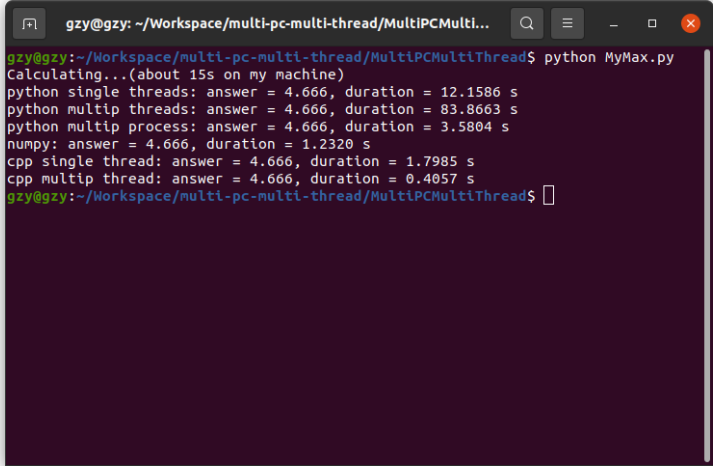
5	
---	--

表 4.6 Python 测试结果

序号	Python 单线程	Python 多线程	Python 多进程	NumPy	调用 C++单线程	调用 C++多线程
1	12.8963	79.4570	3.7461	1.2431	1.8119	0.3227
2	12.0995	80.5321	3.5267	1.2366	1.8074	0.4937
3	13.3200	85.3338	3.6902	1.2441	1.8117	0.4699
4	12.6927	80.0054	3.6851	1.2430	1.8011	0.3260
5	12.1586	83.8663	3.5804	1.2320	1.7985	0.4057
平均	12.4618	80.9652	3.6346	1.2387	1.8047	0.3870
加速比	1.00	0.15	3.43	10.06	6.91	32.20

以上时间单位均为 s，由于第 3 组数据偏离其它数据，所以将其剔除。求得 Python 中单线程平均时间为 12.4618s，多线程平均时间为 80.9652s，多进程平均时间为 1.2387s，NumPy 平均时间为 1.2387s，调用 C++单线程平均时间为 1.8047s，调用 C++多线程平均时间为 0.3870s。加速比分别为 1.00、0.15、3.43、10.06、6.91、32.20。

## 4.3. 结果分析

### 4.3.1. 多机多线程结果分析

在求和 SUM 运算中，单机多线程与多机多线程的加速比均为 5 倍多，且多机多线程慢于单机的多线程。推测原因为由于传输数据的耗时较长，且客户端与服务端的性能差异较大，导致多机时间有所延长，多机加速效果不够明显。

在求最大值 MAX 运算中，单机多线程达到了 4 倍多的加速比，多机多线程达到了 9 倍多的加速比，基本符合预期效果。

在排序 SORT 运算中，我们采用冒泡排序法进行对比试验，其单机多线程与多机多线程的加速比均达到了惊人的 300 多倍。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。使用冒泡排序  $64 \times 200$  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  $1.64e+8/2.56e+6$  倍。单机多线程中，以 4 核 CPU 为例，加速比为  $64 \times 4 = 256$  倍，而实际加速比为 303.93 倍，其原因是由于计算机性能存在差异，而在计算加速比时采用双机平均耗时进行对比，导致加速比有所增加。多机多线程的加速比为 315.38，相比单机多线程提升较少，其原因是由于数据量较大，网络传输需要时间，导致其加速效果不够明显。

#### 4.3.2. Python 调用对比分析

可以看到，Python 单线程耗时十分大，但多线程比单线程耗时更大。这是因为 Python 在执行时需要用到一种叫作全局解释器锁（Global Interpreter Lock, GIL）的机制，相当于一种互斥锁。由于一个 Python 进程只拥有一个 GIL，所以无法利用多核 CPU，由于程序实现的是 CPU 密集型任务，单线程的效率反而比多线程要高。但我们可以通过多进程来解决这个问题，可以看到多进程的速度比单线程快，且加速比达到了 3.43 倍，距离理论上的 4 倍（4 个 CPU 内核同时工作）并不是很远。

使用 NumPy 可以进一步加速运算（加速比为 10.06），相较于调用 C++ 单线程（加速比为 6.91）有足够的优势，但远不如调用 C++ 多线程（加速比为 32.20）。可以看出，从 Python 中调用 C++ 程序是一种不错的提升 Python 运行效率的方法。



## 5. 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) 使用多机多线程、`SSE` 指令集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) 对课程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。



## 6. 重现时的注意事项

可以在 gitee 上直接下载到我们的代码。

[https://gitee.com/guo\\_zhanyu/multi-pc-multi-thread](https://gitee.com/guo_zhanyu/multi-pc-multi-thread)

### 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 args 中，与默认不同的是，需要增加两个参数：“-lpthread”和“-mcmmodel=large”

### 6.2. IP 地址

IP 地址配置位于头文件“mySrvClt.hpp”中，第 83 行左右的位置。

```
1. #ifndef CLIENT
2. #define SERVER_IP "127.0.0.1" // 服务端 IP 地址
3. #endif
```

在服务端使用 ifconfig 命令查询服务端 IP 地址，然后进行修改。

### 6.3. 运算数据量

运算数据量配置位于头文件“mySrvClt.hpp”中，第 87 行左右的位置。

```
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
```

其中每个宏定义的含义如下表所示。

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. 排序数据量

排序数据量配置位于头文件“mySrvClt.hpp”中，第 93 行左右的位置。

```
1. #define S_SUBDATANUM 400 // 减少数据量进行排序
2. #define S_SRV_SUBDATANUM 200 // 单 PC 小数据量测试
3. #define S_CLT_SUBDATANUM 200 // 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. #ifndef SERVER
8. #define S_SRV_DATANUM (S_SRV_SUBDATANUM * MAX_THREADS) // SRV 数据量
9. #endif
```

其中每个宏定义的含义如下表所示。

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，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. 传输数据量

网络传输数据量配置位于头文件“mySrvClt.hpp”中，第 103 行左右的位置。

```
1. #define S_ONCE 1600 // 一次发送 100 个 double
2. #define S_TIMES (S_CLT_DATANUM / S_ONCE) // 总共发送 100 个的次数
3. #define S_LEFT (S_CLT_DATANUM % S_ONCE) // 发送剩余不足的数据
```

其中每个宏定义的含义如下表所示。

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)



## 7. 分工

工作	完成人
多机通信	王渤
多线程	郭展羽
SSE 加速	郭展羽、王渤
报告代码实现部分	郭展羽
报告测试部分	王渤
其余	郭展羽、王渤



## 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] Zhu N, Zhao H. IoT applications in the ecological industry chain from information security and smart city perspectives[J]. Computers & Electrical Engineering, 2018, 65: 34-43.