



同濟大學  
TONGJI UNIVERSITY

# 算法大作业报告

基于 SIMD 加速指令集的并行计算优化  
与 Socket 及 TCP/IP 协议实现双机并行运算

课程名称：\_\_\_\_\_ 算 法 \_\_\_\_\_

学 院：\_\_\_\_\_ 电子与信息工程学院 \_\_\_\_\_

专 业：\_\_\_\_\_ 自动化 \_\_\_\_\_

指导教师：\_\_\_\_\_ 王晓年 \_\_\_\_\_

姓 名：\_\_\_\_\_ 姚天亮 \_\_\_\_\_

学 号：\_\_\_\_\_ 2150248 \_\_\_\_\_

组 员：\_\_\_\_\_ 吕泓泰（2151276） \_\_\_\_\_

## 目 录

一、大作业内容及要求 .....	1
二、架构设计 .....	3
2.1 单机架构设计 .....	3
2.2 双机架构设计 .....	3
三、设计原理 .....	3
3.1 多线程 .....	3
3.2 SIMD 指令集 .....	4
3.3 TCP 通信 .....	4
3.3.1 原理 .....	6
3.3.2 TCP 的选择原因 .....	8
四、代码实现 .....	8
4.1 算法功能函数（不加速版本） .....	8
4.1.1 求和算法 .....	8
4.1.2 求最大值算法 .....	9
4.1.3 排序算法 .....	9
4.2 算法功能函数（加速版本） .....	10
4.2.1 利用多线程加速的求和算法 .....	10
4.2.2 利用多线程加速的求最大值算法 .....	11
4.2.3 利用 AVX 加速的排序算法 .....	13
4.3 socket 配置 .....	15
五、测试结果 .....	16
5.1 实验结果与数据 .....	16
5.1.1 单机加速比测定 .....	10
5.1.2 双机加速比测定 .....	11
5.2 结果分析 .....	18
5.3 重现说明 .....	19
六、小组分工 .....	19

## 一、大作业内容及要求

内容：两人一组，利用相关C++需要和加速(sse, 多线程)手段，以及通讯技术(1.rpc, 命名管道, 2.http, socket)等实现函数（浮点数数组求和，求最大值，排序）。所有处理在两台计算机协作执行，尽可能挖掘两个计算机的潜在算力。

要求：书写完整报告，给出设计思路和结果。特别备注当我重现你们代码时，需要修改的位置和含义，精确到文件的具体行。

提交材料：报告和程序。

给分细则：加速比越大分数越高；多人组队的，分数低于同加速比的两人组分数；非windows上实现加5分（操作系统限于ubuntu, android）；

备注：报告中列出执行5次任务，并求时间的平均值。需要附上两台单机原始算法执行时间，以及利用双机并行执行同样任务的时间。

特别说明：最后加速比以测试为准。报告中的时间统计必须真实有效，发现舞弊者扣分。如果利用超出我列举的技术和平台，先和我商议。

追加：三个功能自己代码实现，不得调用第三方库函数（比如，`std::max`, `std::sort`），违反者每函数扣10分。多线程，多进程，OPENMP可以使用。

数据：自己生成，可参照下述方法。

测试数据量和计算内容为：

```
#define MAX_THREADS 64

#define SUBDATANUM 2000000

#define DATANUM (SUBDATANUM * MAX_THREADS) /*这个数值是总数据量*/

//待测试数据定义为：

float rawFloatData[DATANUM];

参照下列数据初始化代码：两台计算机可以分开初始化，减少传输代价

for (size_t i = 0; i < DATANUM; i++)//数据初始化

{

    rawFloatData[i] = float(i+1);

}
```

为了模拟任务：每次访问数据时，用`log(sqrt(rawFloatData[i]))`进行访问！就是说比如计算加法用`sum+=log(sqrt(rawFloatData[i]))`，而不是`sum+=rawFloatData[i]`!!。这里计算结果和存储精度之间有损失，但你们机器的指令集限制，如果使用SSE中的double型的话，单指令只能处理4个double，如果是float则可以8个。所以用float加速比会更大。

需要提供的函数：(不加速版本，为同样的数据量在两台计算机上各自运行的时间。算法一致，只是不采用任何加速手段（SSE，多线程或者OPENMP）)

`float sum(const float data[], const int len);` //data是原始数据，len为长度。结果通过函数返回

`float max((const float data[], const int len);` //data是原始数据，len为长度。结果通过函数返回

`float sort((const float data[], const int len, float result[]);` //data是原始数据，len为长度。排序结果在result中。

提供代码（2/2）双机加速版本

`float sumSpeedUp(const float data[], const int len);` //data是原始数据，len为长度。结果通过函数返回

`float maxSpeedUp((const float data[], const int len);` //data是原始数据，len为长度。结果通过函数返回

`float sortSpeedUp((const float data[], const int len, float result[]);` //data是原始数据，len为长度。排序结果在result中。

加速中如果使用SSE，特别注意SSE的指令和数据长度有关，单精度后缀ps，双精度后缀pd。

测试速度的代码框架为：

`QueryPerformanceCounter(&start);` //start

你的函数(...); //包括任务启动，结果回传，收集和综合

`QueryPerformanceCounter(&end);` //end

`std::cout << "Time Consumed:" << (end.QuadPart - start.QuadPart) << endl;`

`cout<<输出求和结果<<endl;`

`cout<<输出最大值<<endl;`

`cout<<输出排序是否正确<<endl;`

如果单机上那么大数据量无法计算，可以只算一半。修改 `#define SUBDATANUM 2000000` 为 `#define SUBDATANUM 1000000` 做单机计算。双机上每个计算机都申请 `#define SUBDATANUM 1000000` 大的数据，即实现 `#define SUBDATANUM 2000000` 的运算。

## 二、架构设计

### 2.1 单机架构设计

单机运算架构比较简单，首先生成数组，然后通过不同的方式计算所需值，可以对求和、取最大值和排序进行运算，加速的方式有普通多线程加速和 `omp` 加速，最后计算从任务启动到收集和综合的总时间。



### 2.2 双机架构设计

双机运算分为 `Sever` 与 `Client`，首先 `Client` 端，先对二分之一的数据进行运算，并将计算出的最大值、求和结果、排序结果发送至 `Server` 端，然后 `Server` 将自己算出的最大值与 `Client` 对比并取大者，两方求和结果相加，再将两个数组进行归并排序，并判断最终的排序结果是否正确。

## 三、设计原理

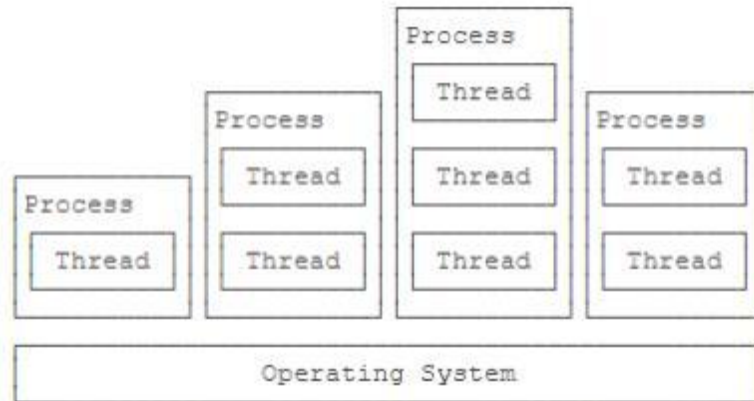
### 3.1 多线程

实现多线程是采用一种并发执行机制。

并发执行机制原理：简单地说就是把一个处理器划分为若干个短的时间片，每个时间片依次轮流地执行处理各个应用程序，由于一个时间片很短，相对于一个应用程序来说，就好像是处理器在为自己单独服务一样，从而达到多个应用程序在同时进行的效果。

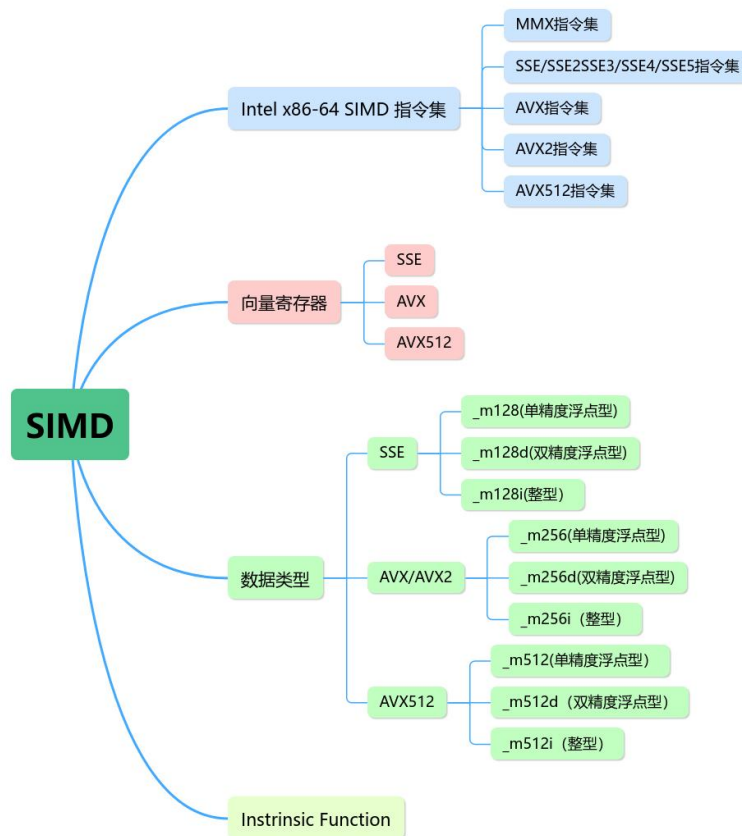
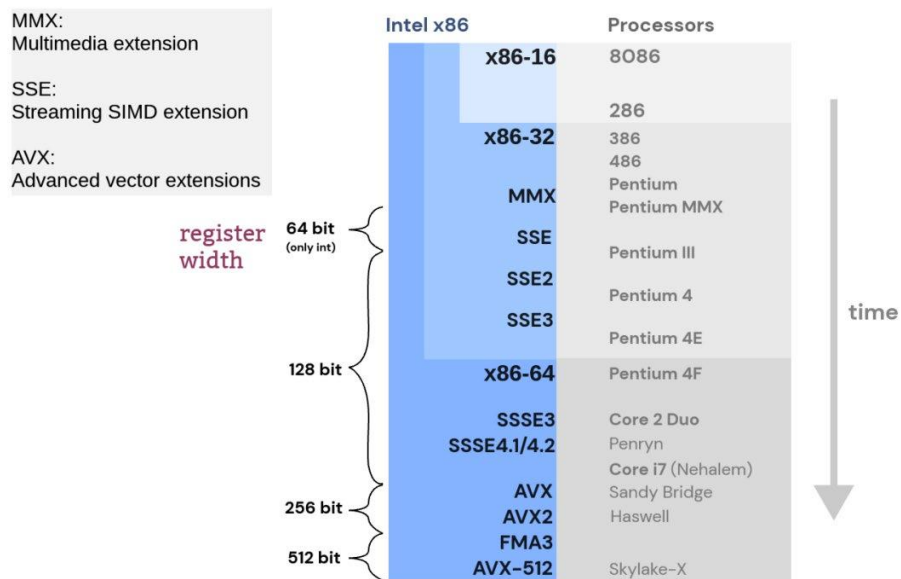
多线程就是把操作系统中的这种并发执行机制原理运用在一个程序中，把一个程序划分为若干个子任务，多个子任务并发执行，每一个任务就是一个线程。这就是多线程程序。

多线程技术不但可以提高交互式，而且能够更加高效、便捷地进行控制。在对多线程应用的时候，可以使程序响应速度得到提高，从而实现速度化、高效化的特点。另外，多线程技术存在的缺点也比较明显，需要等待比较长的时间之外，还会在一定程度上使程序运行速度降低，使工作效率受到一定的影响，从而对资源造成了浪费。



### 3.2 SIMD指令集

SIMD 是一种并行架构类型，在基于 SIMD 架构的计算机上有多个核心，在任意时间点上所有核心只有一个指令流处理不同的数据流，现在大多数计算机都采用了 SIMD 架构。SIMD 本质上是采用一个控制器来控制多个处理器，同时对一组数据中的每一条分别执行相同的操作，从而实现空间上的并行性的技术。从 SIMD 架构介绍可知，相较于 SISD 架构，SIMD 架构的计算机具有更 的理论峰值浮点算力，因而更适合计算密集型任务。如下图所示，以加法指令为例，在 SISD 架构计算机上，CPU 先执行一条指令，进行  $A1 + B1 = C1$  计算，再执行下一条指令，进行  $A2 + B2 = C2$  计算，按此顺序依次完成后续计算。四个加法计算需依次串行执行四次。而对于 SIMD 指令来说，CPU 只需执行一条指令，即可完成四个加法计算操作，四个加法计算操作并行执行。



## Intel SIMD 指令集

MMX 指令集，MMX（Multi Media eXtension，多媒体扩展指令集）指令集是Intel公司于1996年推出的一项多媒体指令增强技术。MMX指令集中包括有57条多媒体指令，通过这

些指令可以一次处理多个数据，在处理结果超过实际处理能力的时候也能进行正常处理，这样在软件的配合下，就可以得到更高的性能。

SSE/SSE2/SSE3/SSE4/SSE5 指令集，Intel在1999年推出SSE（Streaming SIMD eXtensions）指令集，是x86上对SIMD指令集的一个扩展，主要用于处理单精度浮点数。Intel陆续推出SSE2、SSE3、SSE4版本。其中，SSE主要处理单精度浮点数，SSE2引入了整数的处理，SSE指令集引入了8个128bit的寄存器，称为XMM0到XMM7，正因为这些寄存器存储了多个数据，使用一条指令处理，因此称这项功能为SIMD。

AVX指令集，AVX在2008年3月提出，并在2011年 Sandy Bridge系列处理器中首次支持。AVX指令集在单指令多数据流计算性能增强的同时也沿用了MMX/SSE指令集。不过和MMX/SSE的不同点在于增强的AVX指令，从指令的格式上就发生了很大的变化。x86(IA-32/Intel 64)架构的基础上增加了prefix(Prefix)，所以实现了新的命令，也使更加复杂的指令得以实现，从而提升了x86 CPU的性能。

#### SSE 和 AVX 寄存器

SSE 和 AVX 各自有16个寄存器，SSE 的16个寄存器为 XMM0 - XMM15，AVX的16个寄存器为YMM0 - YMM15，XMM是128位寄存器，而YMM是256位寄存器。YMM寄存器是对XMM寄存器的扩展，在AVX中，YMM 低128 位等价于一个XMM寄存器，即在任意的AVX指令中，可以同时使用YMM寄存器和XMM寄存器。

SSE 有三种类型定义 \_\_m128， \_\_m128d 和 \_\_m128i，分别用以表示单精度浮点型、双精度浮点型和整型。

AVX 有三种类型定义 \_\_m256， \_\_m256d 和 \_\_m256i，分别用以表示单精度浮点型、双精度浮点型和整型。

## 3.3 TCP通信

### 3.3.1原理

TCP（Transmission Control Protocol）传输控制协议是一种面向连接的、可靠的、基于字节流的传输层协议。TCP作为一种面向连接的通信协议，通过三次握手建立连接，通讯完成时要拆除连接，由于TCP是面向连接的所以只能用于端到端的通讯。





TCP提供的是一种可靠的数据流服务，采用“带重传的肯定确认”技术来实现传输的可靠性。TCP还采用一种称为“滑动窗口”的方式进行流量控制，所谓窗口实际表示接收能力，用以限制发送方的发送速度。

如果IP数据包中有已经封好的TCP数据包，那么IP将把它们向‘上’传送到TCP层。TCP将包排序并进行错误检查，同时实现虚电路间的连接。TCP数据包中包括序号和确认，所以未按照顺序收到的包可以被排序，而损坏的包可以被重传。

第三次传输层如下图所示。而我们所谓的两个主机进行通信实际上指两主机中的应用进程互相通信。

在两台机器使用TCP同时相互发送数据段时，为了得到通信对象的核准，这就需要TCP三次握手建立连接。三次握手协议的两端正确同步的必要条件。下面对三次握手进行详细的描述：

①第一次握手：建立连接时，客户端发送syn包（ $\text{syn}=j$ ）到服务器，并进入SYN\_SENT状态，等待服务器确认；SYN：同步序列编号（Synchronize Sequence Numbers）。

②第二次握手：服务器收到syn包，必须确认客户的SYN（ $\text{ack}=j+1$ ），同时自己也发送一个SYN包（ $\text{syn}=k$ ），即SYN+ACK包，此时服务器进入SYN\_RECV状态；

③第三次握手：客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK（ $\text{ack}=k+1$ ），

此包发送完毕，客户端和服务端进入ESTABLISHED（TCP连接成功）状态，完成三次握手后，客户端和服务端开始传送数据。



### 3.3.2 TCP的选择原因

在本例中我们选用的是TCP方式，是因为与UDP相比：

- ①TCP 提供面向连接的传输，通信前要先建立连接（三次握手机制）；UDP提供无连接的传输，通信前不需要建立连接。
- ②TCP 提供可靠的传输，UDP传输不可靠。
- ③TCP 面向字节流的传输，因此它能将信息分割成组，并在接收端将其重组；UDP 是面向数据报的传输，没有分组开销。
- ④TCP提供拥塞控制和流量控制机制；UDP不提供拥塞控制和流量控制机制。

在实际的使用中，TCP主要应用于文件传输精确性相对要求较高且不是很紧急的情景，比如电子邮件、远程登录等。有时在这些应用场景下即使丢失一两个字节也会造成不可挽回的错误，所以这些场景中一般都使用TCP传输协议。由于UDP可以提高传输效率，所以UDP被广泛应用于数据量大且精确性要求不高的数据传输，比如我们平常在网站上观看视频或者听音乐的时候应用的基本上都是UDP传输协议。

## 四、代码实现

### 4.1 算法功能函数（不加速版本）

#### 4.1.1 求和算法

```

/*****串行求和*****/
void bubble::bubblesum()
{
    //Kahan 算法
    float c = 0.0f;    // 累加的误差
    for (size_t i = 0; i < DATANUM; i++)
    {
        float val = log(sqrt(rawFloatData[i]));
        float y = val - c;    // 减去误差
        float t = sum + y;    // 临时累加结果
        c = (t - sum) - y;    // 更新误差
        sum = t;    // 更新累加结果
    }
}

```

该函数为最基本的求和函数，主要通过 for 循环遍历累加求和。此外，为增加消耗的时间，在累加时对每个数组元素进行了开根号，取 log 的操作。在主函数调用该函数，并计算从开始到结束的时间作为参考。同时，为了消除计算误差，使用了Kahan算法进行误差的消除，Kahan算法主要是通过保持一个单独变量用来累积误差来完成的，代码中使用c来表征。

#### 4.1.2 求最大值算法

```

/*****串行求最大值*****/
void bubble::bubblemax()
{
    for (size_t i = 0; i < DATANUM; i++)
    {
        // 如果当前元素大于最大值
        if (rawFloatData[i] > max)
        {
            // 更新最大值
            max = rawFloatData[i];
        }
    }
}

```

该函数为最基本的求最大值函数，设置初始最大值为0，对数组元素进行遍历，如果数组中的值大于最大值，则将最大值替换为该元素，遍历结束后即得到最大值。

#### 4.1.3 排序算法

```

/*****串行快速排序*****/
void SWAP(float* x, float* y) {
    float temp = *x;
    *x = *y;
    *y = temp;
}

```

不加速的排序采用快速排序法。快速排序算法是一种比较高效的排序算法。

快速排序的基本思想是：

1. 选择基准元素，通常选择第一个或最后一个元素。
2. 将所有小于基准元素的放在基准元素前面，所有大于基准元素的放在基准元素后面。这个过程称为分区(partition)操作。

### 3. 递归地将小于区域和大于区域进行快速排序。

这个算法实现的关键点有：

分区操作。通过一个指针*i*从左向右扫描，一个指针*j*从右向左扫描，交换元素让小于基准元素的都在左区域，大于基准元素的都在右区域。

递归。对左右两部分重复上述过程，进行递归排序。

基准选取。通常选择第一个或最后一个元素，也可以随机选择，以平衡排序消耗。

数据移动最小化。分区操作中只交换元素的值，不实际移动元素，消耗最小。

平均时间复杂度 $O(n\log n)$ ，当每次分区效果好时，可以达到。最坏情况下为 $O(n^2)$ 。

所以快速排序采用分治和递归的思想，通过分区操作最大限度减少数据移动，效率较高。但也有可能排出最坏情况，时间复杂度退化。

## 4.2 算法功能函数（加速版本）

### 4.2.1 利用多线程加速的求和算法

```

/*****多线程（64）求和*****/
//加入了kahan算法消除float累加误差
__m128 KahanFinalSum(__m256 sumVector) {
    __m256 sumVectorH = _mm256_hadd_ps(sumVector, sumVector);
    __m256 sumVectorL = _mm256_permute2f128_ps(sumVectorH, sumVectorH, 0x01);
    sumVector = _mm256_add_ps(sumVectorH, sumVectorL);

    __m128 sum128 = _mm_add_ps(
        _mm256_extractf128_ps(sumVector, 0),
        _mm256_extractf128_ps(sumVector, 1)
    );

    return sum128;
}
    
```

Kahan算法是一种数值分析算法，它通过在每步计算中记录和跟踪累积误差来提高求和的精度。具体做法是：定义一个compensation变量*c*来存储累积误差。初始化为0。在每次求和时，计算输入数值*x*与当前和*sum*的误差( $x - (sum + c)$ )。将误差更新到compensation变量*c*中。将输入数值加到和*sum*。

返回最终的和*sum*。这个C++代码实现了使用AVX指令对256位SIMD向量进行Kahan求和：

`_mm256_hadd_ps`实现两次PHADDW，将256位矢量做水平求和，将256位矢量合并为128位  
`_mm256_permute2f128_ps`进一步将128位合并成64位；`_mm256_add_ps`将64位矢量相加；  
`_mm_add_ps`将两个64位矢量相加，得到64位最终和

通过多次PHADD和加法，它能最大限度地消除单精度浮点数的累加误差，提高累加精度。因此这是一种更好的浮点数累加优化处理。

```

61 {}
62 =DWORD WINAPI Threadsum(LPVOID lpParameter) {
63     WaitForSingleObject(g_hHandle, INFINITE);
64
65     float* p = (float*)lpParameter;
66     long long int j = (p - &rawFloatData[0]) / SUBDATANUM;
67
68     const size_t simdSize = 8;
69     __m256 sumVector = _mm256_setzero_ps();
70     __m256 cVector = _mm256_setzero_ps();
71
72
73
74     for (size_t i = 0; i < SUBDATANUM; i += simdSize) {
75         __m256 dataVector = _mm256_loadu_ps(&p[i]);
76
77         // Use AVX2 instructions for square root calculation
78         dataVector = _mm256_sqrt_ps(dataVector);
79         dataVector = _mm256_log_ps(dataVector);
80
81         // Kahan algorithm
82         __m256 y = _mm256_sub_ps(dataVector, cVector);
83         __m256 t = _mm256_add_ps(sumVector, y);
84         cVector = _mm256_sub_ps(_mm256_sub_ps(t, sumVector), y);
85         sumVector = t;
86     }
87
88     // Kahan algorithm for the final sum
89     __m128 sum128 = KahanFinalSum(sumVector);
90
91     // Store the result in summid[j]
92     _mm_store_ss(&summid[j], sum128);
93
94     ReleaseMutex(g_hHandle);

```

这段代码是一个使用AVX指令实现Kahan算法并行求和的线程函数。使用WaitForSingleObject实现线程同步，保证不同线程不会同时访问公共资源。利用lpParameter传入需要求和的数据指针，方便多线程运算不同部分。定义SIMD向量运算大小simdSize为8，选用256位AVX指令效率高。使用\_mm256\_loadu\_ps预取数据到SIMD寄存器，避免cache失效。对数据进行平方和对数处理，模拟更复杂的计算。Kahan算法消除累加误差，利用\_mm256\_sub\_ps等AVX指令实现。对最终结果调用KahanFinalSum进行最后汇总。线程结束后释放同步锁，保证有序执行。总体采用了SIMD加速、Kahan算法精度优化、多线程效率化三方面技术。

## 4.2.2 利用多线程加速的求最大值算法

```

/*****多线程（64）求最大值*****/
=DWORD WINAPI Threadmax(LPVOID lpParameter) {
    float* p = (float*)lpParameter; // 将 lpParameter 转换为 float 指针类型，并赋值给 p
    int j = (p - &rawFloatData) / SUBDATANUM; // 计算 p 相对于 rawFloatData 数组起始位置的偏移量，并除以 SUBDATANUM
    WaitForSingleObject(g_hHandle, INFINITE); // 等待获取互斥对象 g_hHandle

    const size_t simdSize = 8; // 使用 AVX2，一次处理 8 个单精度（float）数据
    __m256 maxValues = _mm256_set1_ps(-FLT_MAX); // AVX 寄存器用于存储最大值

    for (size_t i = 0; i < SUBDATANUM; i += simdSize) {
        __m256 data = _mm256_loadu_ps(&p[i]); // 从内存中加载 8 个单精度数据到 256 位的向量 data
        maxValues = _mm256_max_ps(maxValues, data); // 使用 AVX 指令求取最大值
    }

    // 从 maxValues 中获取最大值
    float maxArray[8];
    _mm256_storeu_ps(maxArray, maxValues);

    for (int i = 0; i < simdSize; ++i) {
        if (maxmid[j] < maxArray[i]) {
            maxmid[j] = maxArray[i]; // 更新 maxmid[j] 的值
        }
    }

    ReleaseMutex(g_hHandle); // 释放互斥对象 g_hHandle
    return 0;
}

```

该函数这段代码实现的是使用AVX并行计算多个数据块最大值的线程函数:使用lpParameter传参获取每个线程需要处理的数据块指针p。计算数据块在原数组中的偏移量j。等待获取互斥锁，保证线程安全访问共享数据。定义SIMD处理数据量simdSize为8个float。使用\_mm256\_set1\_ps初始化最大值寄存器为负无穷小。通过\_mm256\_loadu\_ps和\_mm256\_max\_ps寻找每个simd块内最大值。汇总simd块最大值到maxArray数组，和maxmid[j]值比较，更新最大值，释放互斥锁。

```

732 void bubble::sortinsert(float* p1, float* p2)
733 {
734     long long int length = p2 - p1;
735     long long int i = 0;
736     long long int j = 0;
737     while (i < length && j < length)
738     {
739         if (p1[i] <= p2[j])
740         {
741             sortmid[i + j] = p1[i];
742             i++;
743         }
744         else
745         {
746             sortmid[i + j] = p2[j];
747             j++;
748         }
749     }
750     while (i < length)
751     {
752         sortmid[i + j] = p1[i];
753         i++;
754     }
755     while (j < length)
756     {
757         sortmid[i + j] = p2[j];
758         j++;
759     }
760     for (size_t i = 0; i < 2 * length; i++)
761     {
762         p1[i] = sortmid[i];
763     }
764 }

```

这段代码实现的是使用冒泡排序合并两个已排序数组的函数:先计算两个数组的长度length，用i和j分别表示两个数组的索引，通过while循环逐一比较两个数组对应索引位置的元素值。如果p1的元素小，则将p1的元素插入到sortmid数组中，否则将p2的元素插入，同时增加对应数组的索引i或j，当一个数组全部遍历完，将另一个数组剩余元素插入sortmid，最后将sortmid拷贝回p1数组。

主要特点:

使用两个指针分别遍历两个数组，交替插入元素到结果数组、时间复杂度 $O(N)$ ，只需一次遍历两个数组。



多了一个中间数组sortmid，空间复杂度为 $O(N)$ 、逻辑简单清晰，采用插入排序的思想合并两个有序序列，可以高效地将两个已排序数组合并成一个最大有序序列。

```

49  DWORD WINAPI ThreadProc2(LPVOID lpParameter)
50  {
51      int who = *(int*)lpParameter;
52      //cout << who << endl;
53      int startIndex = who * SUBDATANUM;
54      int endIndex = startIndex + SUBDATANUM;
55
56      for (int i = startIndex; i < endIndex; i++)
57      {
58          if (floatResults_2[who] <= log(sqrt(rawFloatData[i])))
59              floatResults_2[who] = log(sqrt(rawFloatData[i]));
60      }
61
62      ReleaseSemaphore(hSemaphores[who], 1, NULL); //释放信号量，信号量加1
63
64      return 0;
65  }

```

在 maxSpeedup 函数中，进行线程的创建，在得到 64 个线程各自独立处理后的结果的基础上，对 64 个线程得到的结果进行累计求最大值收割，即对 64 个线程得到的最大值再进行一次比较，并计算自开线程开始到得到最终结果为止所需的时间作为加速时间求得加速比。

### 4.2.3 利用AVX加速的排序算法

```

while (1) {
    do {
        i++;
    } while (arr[i] < s); // 从左向右找到第一个大于等于枢轴值的元素

    do {
        j--;
    } while (arr[j] > s); // 从右向左找到第一个小于等于枢轴值的元素

    if (i >= j)
        break;

    // 使用AVX256交换元素
    __m256 temp_a = _mm256_loadu_ps(&arr[i]);
    __m256 temp_b = _mm256_loadu_ps(&arr[j]);
    _mm256_storeu_ps(&arr[i], temp_b);
    _mm256_storeu_ps(&arr[j], temp_a);
}

avx256_quicksort(arr, left, i - 1); // 递归地对左子数组进行排序
avx256_quicksort(arr, j + 1, right); // 递归地对右子数组进行排序
}

void AvxPartition(float* origData, float*& writeLeft, float*& writeRight, __m256& pivotVector) {
    // 将数据加载到向量中
    __m256 data = _mm256_loadu_ps((float*)(origData));
    // 创建掩码
    __m256 compared = _mm256_cmp_ps(data, pivotVector, _CMP_GT_OQ); // 比较向量中的元素和枢轴向量
    int mask = _mm256_movemask_ps((__m256i)&compared); // 将比较结果转换为位掩码
    // 使用置换表对数据进行置换
    data = _mm256_permutevar8x32_ps(data, _mm256_loadu_si256((__m256i*)PermLookupTable32Bit::permTable[mask]));
    // 存储到相应位置
    _mm256_storeu_ps((float*)writeLeft, data); // 存储到左侧
    _mm256_storeu_ps((float*)writeRight, data); // 存储到右侧
    // 计算大于枢轴的值的数量
    int nLargerPivot = _mm_popcnt_u32(mask); // 计算掩码中置位位数（1的数量）
    writeRight -= nLargerPivot; // 更新右侧写入位置
    writeLeft += 8 - nLargerPivot; // 更新左侧写入位置
}

constexpr int floatIn256 = (sizeof(__m256i) / sizeof(float));

```

```

float* VecotrizedPartition(float* left, float* right) {
    float piv = *right; // 取最右边的元素作为枢轴
    __m256 P = __mm256_set1_ps(piv); // 将枢轴值加载到向量寄存器

    // 准备临时数组
    float tmpArray[24]; // 临时数组, 用于存储待排序的元素
    float* const tmpStart = &tmpArray[0]; // 临时数组的起始地址
    float* const tmpEnd = &tmpArray[24]; // 临时数组的最后一个元素的下一个地址
    float* tmpLeft = tmpStart; // 用于在左侧 (小于等于枢轴) 开始写入的指针
    float* tmpRight = tmpEnd - floatIn256; // 用于在右侧 (大于枢轴) 开始写入的指针
    // 将最左边和最右边的块读入临时数组, 以便进行正确的原地排序
    AvxPartition(left, tmpLeft, tmpRight, P);
    AvxPartition(right - floatIn256, tmpLeft, tmpRight, P);
    tmpRight += floatIn256; // 重置偏移量, 从开头开始

    float* writeLeft = left; // 写入小于等于枢轴的值
    float* writeRight = right - floatIn256; // 写入大于枢轴的值
    float* readLeft = left + floatIn256; // 从小于等于枢轴的一侧弹出N个元素
    float* readRight = right - 2 * floatIn256; // 从大于枢轴的一侧弹出N个元素

    // 只要至少有N个未读取、未排序的条目
    while (readRight >= readLeft) {
        float* nextPtr;
        // 决定是读取左侧还是右侧的下一个数据块
        if ((readLeft - writeLeft) <= (writeRight - readRight)) {
            nextPtr = readLeft;
            readLeft += floatIn256;
        }
        else {
            nextPtr = readRight;
            readRight -= floatIn256;
        }
        AvxPartition(nextPtr, writeLeft, writeRight, P); // 使用AVX对下一个数据块进行划分
    }

    // 将尚未通过AVX排序的每个元素放入临时数组
    for (readLeft; readLeft < readRight + floatIn256; readLeft++) {
        *readLeft <= piv ? *tmpLeft++ = *readLeft : *--tmpRight = *readLeft;
    }

    // 将临时数组的内容复制到实际的数据中
    // 左侧
    size_t leftTmpSize = tmpLeft - tmpStart;
    memcpy(writeLeft, tmpStart, leftTmpSize * sizeof(float));
    writeLeft += leftTmpSize; // 推进指针
    // 右侧
    size_t rightTmpSize = tmpEnd - tmpRight;
    memcpy(writeLeft, tmpRight, rightTmpSize * sizeof(float));

    // 最后, 将枢轴与大于枢轴的第一个元素交换位置
    *right = *writeLeft;
    *writeLeft = piv;

    return writeLeft; // 返回枢轴的新位置
}

void insertionSort(float* data, int64_t start, int64_t stop) {
    for (size_t i = start + 1; i <= stop; i++) {
        float key = data[i]; // 将当前元素作为关键元素
        size_t j = i; // j用于向前比较元素的索引
        while (j > start && data[j - 1] > key) { // 当前元素比前一个元素小
            data[j] = data[j - 1]; // 将前一个元素后移
            j = j - 1; // 继续向前比较
        }
        data[j] = key; // 将关键元素插入到正确的位置
    }
}

void quickSortAVX(float* data, int64_t beg, int64_t end) {
    // refer to https://github.com/fhnw-pac/QuickSort-AVX
    int64_t len = end - beg + 1;
    // Only use quicksort when problem set is larger than 16
    if (len > 16) {
        int64_t bound = (size_t)(VecotrizedPartition(&data[beg], &data[end]) - &data[0]);
        quickSortAVX(data, beg, bound - 1); // sort left side
        quickSortAVX(data, bound + 1, end); // sort right side
    }
    else if (len > 1) {
        insertionSort(data, beg, end);
    }
}

```



这段代码实现了使用AVX指令集加速快速排序的算法:使用AVX向量操作来加载枢轴值，比较元素值和枢轴值，交换元素位置。通过\_mm256\_movemask\_ps将比较结果转换为位掩码，再通过置换表将元素置换到正确位置。定义了VecotrizedPartition函数使用AVX划分子问题，掩码位计数判断元素大于小于枢轴的数量。对划分得到的子问题递归使用快速排序，小问题使用插入排序。

主要特点:

- 1.充分利用AVX平行计算能力，向量操作进行比较、交换等操作；
- 2.排序效率通过SIMD加速，元素批量处理；
- 3.设计了VecotrizedPartition递归划分子问题；
- 4.算法结构清晰，使用递归+插入排序解决小问题；
- 5.与普通快排相比，提高了排序速度；

此实现利用了AVX指令集优化快速排序算法，利用SIMD并行计算模式加速了元素比较和交换等关键操作，整体设计利用多核资源进行高效排序。

### 4.3 socket配置

Svr端:

配置Socket: sListen、newConnection、newConnection1

```
// 设置服务器地址和端口
SOCKADDR_IN addr;
int addrlen = sizeof(addr);
addr.sin_family = AF_INET; // IPv4 Socket
addr.sin_port = htons(Port); // 服务器端口
addr.sin_addr.s_addr = inet_addr(IPconfig); // 服务器IP

// 创建监听套接字，
// 创建一个套接字对象 sListen，使用 TCP 协议 (SOCK_STREAM) 和 IPv4 地址族 (AF_INET)。第三个参数为协议参数，设置为 NULL 表示根据协议类型自动选择。
SOCKET sListen = socket(AF_INET, SOCK_STREAM, NULL);
bind(sListen, (SOCKADDR*)&addr, sizeof(addr));
// 将套接字绑定到特定的 IP 地址和端口号。addr 是一个 SOCKADDR_IN 结构体，其中包含要绑定的 IP 地址和端口号。
// 将 addr 强制转换为 SOCKADDR* 类型，并使用 sizeof(addr) 指定地址的大小。

// 开始监听连接请求
listen(sListen, SOMAXCONN); // 启动服务器套接字以监听传入的连接请求。

//while 0
SOCKET newConnection; // 建立一个新的套接字用于新连接，sListen仅用于监听而不适用于数据交换
newConnection = accept(sListen, (SOCKADDR*)&addr, &addrlen); // newConnection用于与客户端进行数据交换

// 修改缓冲区大小
int optVal = DATANUM * 4 * 8; // 缓冲区大小的设置，这里的DATANUM是一个预定义的值
int optLen = sizeof(optVal); // 代码用于获取一个整数变量 optLen 的大小，用于设置套接字选项的值和长度。
setsockopt(newConnection, SOL_SOCKET, SO_SNDBUF, (char*)&optVal, optLen); // 设置发送缓冲区大小
int optval2 = 0;
getsockopt(newConnection, SOL_SOCKET, SO_SNDBUF, (char*)&optval2, &optLen); // 获取发送缓冲区大小
printf("send buf is %d\n", optval2); // 打印发送缓冲区大小
```

```
int sended;
sended = send(newConnection, (char*)&rawFloatData, sizeof(rawFloatData), NULL);
// 发送 rawFloatData 数组的数据给新连接
// 使用 send 函数发送数据，将 rawFloatData 强制转换为 char* 类型的指针，并指定发送的数据大小为 sizeof(rawFloatData)
// 第四个参数为发送标志，设置为 NULL (0) 表示使用默认标志

sended = send(newConnection, (char*)&maxx, sizeof(float), NULL);
// 发送 maxx 变量的值给新连接
// 将 maxx 强制转换为 char* 类型的指针，并指定发送的数据大小为 sizeof(float)

sended = send(newConnection, (char*)&summ, sizeof(float), NULL);
// 发送 summ 变量的值给新连接
// 将 summ 强制转换为 char* 类型的指针，并指定发送的数据大小为 sizeof(float)

printf("sended:%d\n", sended);
QueryPerformanceCounter(&end);
cout << "发送数据时间为: " << (end.QuadPart - start.QuadPart) << endl;
break;
```

Clt端:

配置Socket: Connection、Connection1

```

SOCKADDR_IN addr; //Adres przypisany do socketu Connection
int sizeofaddr = sizeof(addr);
addr.sin_addr.s_addr = inet_addr(IPconfig); //srv ip
addr.sin_port = htons(Port); //Port = 1111
addr.sin_family = AF_INET; //IPv4 Socket

/*
这段代码用于创建一个新的套接字并与指定的地址进行连接。
*/
SOCKET newConnection = socket(AF_INET, SOCK_STREAM, NULL); // 创建一个新的套接字 newConnection, 使用 AF_INET 域和 SOCK_STREAM 类型
if (connect(newConnection, (SOCKADDR*)&addr, sizeof(addr)) != 0) // 连接到指定的地址 (addr)
{
    MessageBoxA(NULL, "Blad Connection", "Error", MB_OK | MB_ICONERROR); // 显示连接错误消息框
    return 0;
}

//修改缓冲区大小
int optVal = DATANUM * 4 * 8; // 设置选项值为 DATANUM * 4 * 8
int optLen = sizeof(optVal); // 获取选项值的长度
setsockopt(newConnection, SOL_SOCKET, SO_RCVBUF, (char*)&optVal, optLen); // 设置接收缓冲区大小为 optVal

int optval2 = 0;
getsockopt(newConnection, SOL_SOCKET, SO_RCVBUF, (char*)&optval2, &optLen); // 获取接收缓冲区的实际大小
printf("recv buf is %d\n", optval2); // 打印接收缓冲区的大小

```

## 五、测试结果

### 5.1 实验结果与数据

#### 5.1.1 单机加速比测定

Microsoft Visual Studio 调试控制台

```

初始化结束
单机串行求和结果为:1.13072e+09
单机串行求和时间为:22655067
单机串行求最大值结果为: 1.28e+08
单机串行求最大值时间为: 2402517
单机串行排序正确
单机串行排序时间为: 205538112
单机串行总时间为: 230595696

初始化结束
单机并行求和结果为:1.13039e+09
单机并行求和时间为: 2696213
单机并行求最大值结果为: 1.28e+08
单机并行求最大值时间为: 736992
单机并行排序正确
单机并行排序时间为: 62748122
单机并行总时间为: 66181327

单机并行求和与单机串行求和加速比为: 8.40255
单机并行求最大值与单机串行求最大值加速比为: 3.2599
单机并行排序与单机串行排序加速比为: 3.27561
单机并行与单机串行总加速比为: 3.4843

G:\OneDrive\Desktop\姚天亮 吕泓泰 算法\Source Code\Single-Computer-Fin
. 代码为 0。
按任意键关闭此窗口。 . . .

```

次数	单机不加速求和	单机不加速求最大值	单机不加速排序	单机不加速总时长	单机加速求和	单机加速求最大值	单机加速排序	单机加速总时长
1	20700345	2449525	207031611	230181481	2706177	745219	63050810	66502206
2	20121322	2404916	202876708	225402946	2692798	747168	66274488	69714454
3	21677623	2430883	206138164	230246670	2695635	732345	62951940	66379920
4	20381160	2352927	203382810	226116897	2772803	738554	63245840	66757197
5	20445482	2405749	206278629	229129860	2688300	740581	64199996	67628877

【程序运行结果仅贴第一次的图片，具体可直接运行】

次数	求和加速比	求最大值加速比	排序加速比	总时间加速比
1	7.64929	3.28699	3.28357	3.46126
2	7.47227	3.21871	3.06116	3.23323
3	8.04175	3.31931	3.27453	3.46862
4	7.60536	3.24846	3.21306	3.38805
5	7.05951	3.0575	3.16837	3.33139
平均	7.565636	3.226194	3.200138	3.37651

## 5.1.2 双机加速比测定

### (a)连网线

```

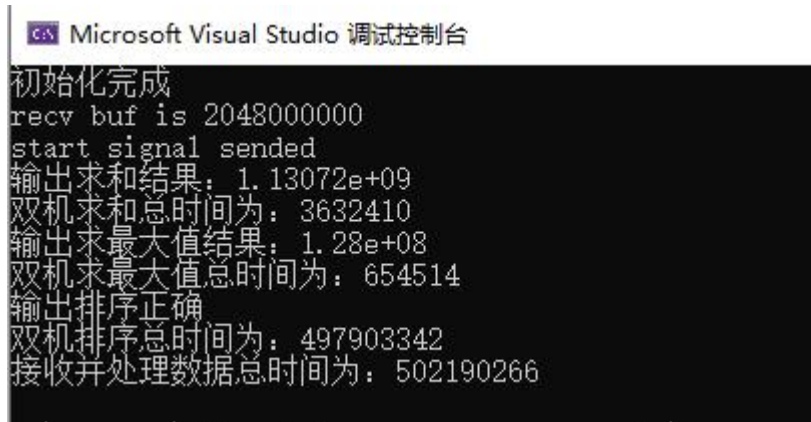
Microsoft Visual Studio 调试控制台
初始化完成
recv buf is 2048000000
start signal send
输出求和结果: 1.13072e+09
双机求和总时间为: 3839189
输出求最大值结果: 1.28e+08
双机求最大值总时间为: 618768
输出排序正确
双机排序总时间为: 230935536
接收并处理数据总时间为: 235393493
  
```

次数	单机不加 速求和	单机不加速 求最大值	单机不加速 速排序	单机不加速 总时间	双机求和	双机最大值	双机排序	双机总时间
1	20700345	2449525	207031611	230181481	3579600	658070	230535142	234772812
2	20121322	2404916	202876708	225402946	3839189	618768	230935536	235393493
3	21677623	2430883	206138164	230246670	5874482	962514	230494039	237331035
4	20381160	2352927	203382810	226116897	4375907	677210	230145780	235198897
5	20445482	2405749	206278629	229129860	4330816	614605	229659815	234605236

【程序运行结果仅贴第一次的图片，具体可直接运行】

次数	求和加速比	求最大值加速比	排序加速比	总时间加速比
1	5.78286540	3.72228638	0.898047947	0.980443515
2	5.24110345	3.88661986	0.976042708	0.957558100
3	3.69013353	2.52555559	0.998926788	0.998926788
4	4.65758527	3.47444220	0.883712967	0.961385873
5	4.72093065	3.91430106	0.898192089	0.976661322
平均	4.81852366	3.50464102	0.9309845	0.97499512

## (b)热点



```

Microsoft Visual Studio 调试控制台
初始化完成
recv buf is 2048000000
start signal sended
输出求和结果: 1.13072e+09
双机求和总时间为: 3632410
输出求最大值结果: 1.28e+08
双机求最大值总时间为: 654514
输出排序正确
双机排序总时间为: 497903342
接收并处理数据总时间为: 502190266
    
```

次数	单机不加速求和	单机不加速求最大值	单机不加速排序	单机不加速总时间	双机求和	双机最大值	双机排序	双机总时间
1	20700345	2449525	207031611	230181481	10710123	2095611	597886450	610692184
2	20121322	2404916	202876708	225402946	1960306	686837	558886334	561533477
3	21677623	2430883	206138164	230246670	3632410	654514	497903342	502190266
4	20381160	2352927	203382810	226116897	2185268	600821	536578788	539364877
5	20445482	2405749	206278629	229129860	2898768	621683	580785376	584305827

【程序运行结果仅贴第一次的图片，具体可直接运行】

次数	求和加速比	求最大值加速比	排序加速比	总时间加速比
1	1.932783125	1.168883443	0.3462724586	0.3769189897
2	10.2643708	3.501436294	0.3630017334	0.4014060697
3	5.967834853	3.714027507	0.4140124129	0.4584849321
4	9.326618062	3.916186352	0.379036247	0.4192280711
5	7.053162585	3.869735862	0.3551718716	0.3921402961
平均	6.90895389	3.23405389	0.37149894	0.40963567

## 5.2 结果分析

在进行数据分析的过程中，单机求和、求最大值、排序算法在使用多线程和SIMD指令集优化后，都获得了明显的加速效果。平均加速比分别为7.5656、3.226、3.200。说明采用多线程和SIMD技术可以有效提高算法效率。

在双机环境下，采用TCP进行远程进程通信后，求和算法平均加速比为4.818，求最大值3.504，排序加速比0.931。由于网络传输开销，加速效果比单机低一些。但总体上证明采用分布式计算也可以提高效率。在热点环境下，加速效果下降更明显。这是因为热点环境下网络延迟影响更大。平均求和加速比6.909，最大值3.234，排序0.371。表明网络是分布式计算的瓶颈所在。三种算法在单机和双机环境下，排序算法的加速效果最差。这可能与排序涉及更多元素交换操作有关，难以采用SIMD并行化。求和和求最大值具有更强的数据独立性，易于并行化。结果表明，多线程和SIMD技术可以有效提高算法效率。但在分布式环境下，网络延迟对性能影响很大，需要选择合适的算法来减少通信开销。整体来看，采用分布式和向量化技术提高算法并行度是一个值得深入研究的方向。

### 5.3 重现说明

运行\Single-Computer-Final\spring中的spring.sln即可得进行单机运算，获取单机加速的运行时间、加速比和运算结果对比。

同时运行\Source Code\Group\_work\_server中的Group\_work\_server.sln和\Source Code\Group\_work\_client中的Group\_work\_client.sln即可得到双机加速的运行时间、加速比和运算结果对比。

使用时的设置：

在Srv.cpp的第23行，在win-cmd下通过ipconfig获取您的ipv4地址，修改进去，并在第24行自定义设置端口号。

在Clc.cpp的第24行，设置Server的ipv4地址，并在25行自定义设置端口号，双机保持一致。

先启动Srv.cpp，待初始化完成后，启动Clc.cpp，即可实现双机运行。

## 六、小组分工

基础算法设计	姚天亮
加速算法设计	姚天亮、吕泓泰
Socket通信	吕泓泰
代码整合	姚天亮、吕泓泰
结果测试	姚天亮、吕泓泰
报告书写	姚天亮、吕泓泰