

目录

一、 作业要求及总体思路	3
1.1 内容	3
1.2 总体思路	3
二、 软硬件环境介绍	3
2.1 电脑配置	3
2.2 Visual studio 版本及配置	4
三、 单机版关键算法和代码介绍	5
3.1. 单机不加速版	5
3.1.1 sum 函数	6
3.1.2 max 函数	6
3.1.3 sort 函数	7
3.1.4 不加速单机输出结果（计算机一）	8
3.2. 加速版	8
3.2.1 sum 函数	8
(1) 并行化:	8
(2) AVX 加速:	8
3.2.2 max 函数	9
(1) 并行化:	9
(2) AVX 加速:	9
(3) 局部结果处理:	9
(4) 汇总线程结果:	10
3.2.3 sort 函数	10
(1) 更好的缓存利用率	11
(2) 数据局部性改善	11
(1) 减少了资源瓶颈	11
(2) 减少了缓存污染	11
(1) 减少内存访问开销	12
(1) 更好的指令调度	12
(1) 动态频率调整	12
四、 双机版	12
4.1 运行环境	12
4.2 程序架构思路	13
6. 检查排序	13
4.3 完整建构设计:	13
4.4 关键代码及其分析	14
4.4.1 tcp 面向连接“三次握手”	14
(1) 客户端（Client）:	14
(2) 服务器端（Server）:	15
4.4.2 传输最大值 和 排序结果。	16
(1) 服务器端（Server）:	16
(2) 客户端（Client）:	17
4.5 如何保证数据完整性和不丢包:	19
4.6 输出结果展示	19

五、 结果对比分析	19
5.1 单机不加速版（单机数据量 64*2000000）：	20
sort 函数	20
5.2 单机加速版（单机数据量 64*2000000）：	21
sort 函数	21
5.3 双机加速版不重排序（单机数据量 64*1000000）：	22
sort 函数	22
5.4 双机加速版重排序（单机数据量 64*1000000）：	23
sort 函数	23
5.5 加速比统计：	23
六、 总结和心得	24
6.1 收获	24
6.2 仍可改进的方向	24
七、 分工	25

一、作业要求及总体思路

1.1 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）。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。

1.2 总体思路

把整个项目拆解成四个程序，包括单机不加速版、单机加速版和双机 TCP 通讯的客户端和服务端，其中单机不加速版用于给后续加速算法提供结果和时间基准。

二、软硬件环境介绍

2.1 电脑配置

计算机一：惠普星 14pro

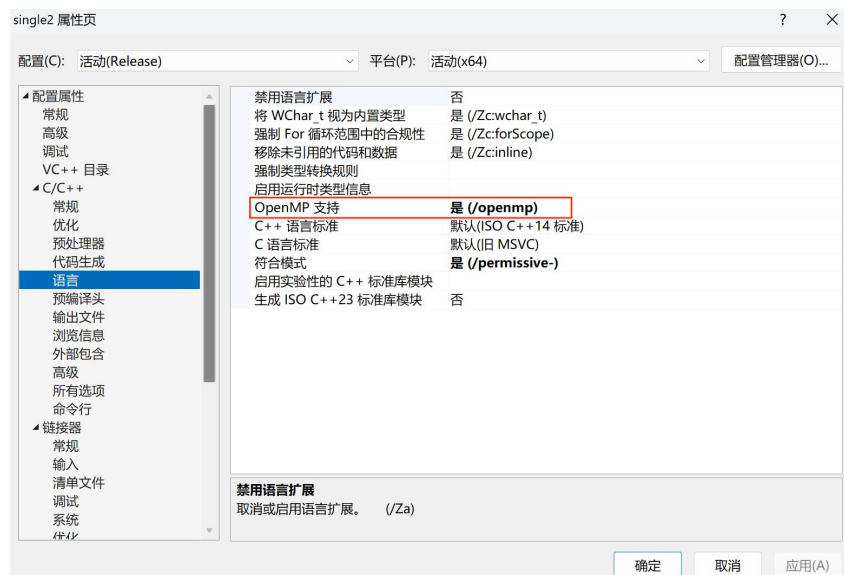
① 设备规格	复制	^
设备名称	summer	
处理器	12th Gen Intel(R) Core(TM) i7-1255U 1.70 GHz	
机带 RAM	16.0 GB (15.7 GB 可用)	
设备 ID	459B2DC4-0830-4CA8-BD2C-B56B17996A63	
产品 ID	00342-30648-44414-AAOEM	
系统类型	64 位操作系统, 基于 x64 的处理器	
笔和触控	为 2 触摸点提供笔和触控支持	

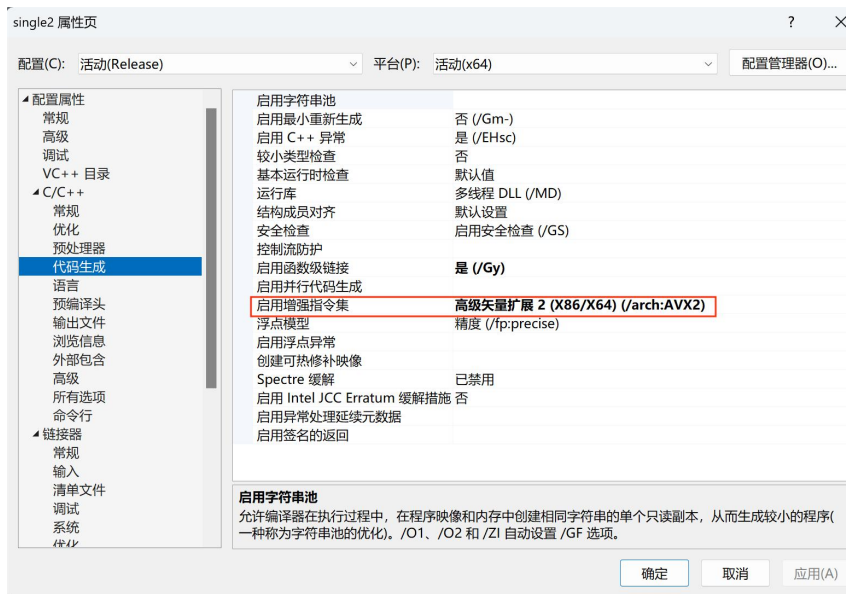
计算机二：DELL G16 i7

设备规格		复制	^
设备名称	DESKTOP-AVK87US		
处理器	12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz		
机带 RAM	16.0 GB (15.7 GB 可用)		
设备 ID	7A02485A-1369-4065-89CF-0B7AD104D155		
产品 ID	00342-30645-57019-AAOEM		
系统类型	64 位操作系统, 基于 x64 的处理器		
笔和触控	没有可用于此显示器的笔或触控输入		

2.2 Visual studio 版本及配置

使用 Visual studio2022，代码涉及 omp 和 AVX，相关设置如下图所示：





三、单机版关键算法和代码介绍

单机版测试数据及初始化均如下：

```
#define MAX_THREADS 64 // 最大线程数
#define SUBDATANUM 2000000 // 每个线程处理的数据量
#define DATANUM (SUBDATANUM * MAX_THREADS) // 总数据量

// 求和和最大值时使用的线程数
#define THREADS_NUM 64

// 待测试数据
float rawFloatData[DATANUM]; // 原始浮点数据数组
// 求最大值时的中间数据
float a[DATANUM];
```

```
int main() {
    // 数据初始化
    for (size_t i = 0; i < DATANUM; i++) {
        //rawFloatData[i] = float(i + 1);
        rawFloatData[i] = log(sqrt(float(i + 1)));
    }
}
```

3.1. 单机不加速版

3.1.1 sum 函数

直接通过一个简单的 for 循环遍历数组中的每个元素，将其逐一累加到一个 double 类型的变量中，最后再转成 float 型。

选择数据类型时，考虑到 float 和 double 的精度区别，分别用两种类型计算了结果，



```
// 求和函数 (float版本)
float mysum_float(const float data[], const int len) {
    float sum = 0.0; // 使用double累加, 避免精度丢失
    for (size_t i = 0; i < len; i++) {
        sum += data[i];
    }
    return static_cast<float>(sum);
}

// 求和函数 (double版本)
double mysum_double(const double data[], const int len) {
    double sum = 0.0;
    for (size_t i = 0; i < len; i++) {
        sum += data[i];
    }
    return sum;
}

// 初始化float数组
for (size_t i = 0; i < DATANUM; i++) {
    rawFloatData[i] = log(sqrt(float(i + 1)));
}

// 初始化double数组
for (size_t i = 0; i < DATANUM; i++) {
    rawDoubleData[i] = log(sqrt(double(i + 1)));
}

// 计算float数组的和
float floatSum = mysum_float(rawFloatData, DATANUM);
double doubleSum = mysum_double(rawDoubleData, DATANUM);
// 禁用科学计数法输出
cout.setf(ios::fixed);
cout << fixed << setprecision(15);
cout << "Float型求和结果: " << floatSum << endl;
cout << fixed << setprecision(15);
cout << "Double型求和结果: " << doubleSum << endl;
```

把 double 型得到的结果作为真值对比，发现误差很大：

```
Float型求和结果: 268435456.0000000000000000
Double型求和结果: 1130722617.726932764053345

C:\Users\张芷溪\Desktop\最终版(2)\single test\x64\Debug
\singletest.exe (进程 9828)已退出, 代码为 0 (0x0)。
按任意键关闭此窗口...
```

但如果将上图中的中间变量声明为 double sum = 0.0，最后再截断为 float，可以有效减少运算过程中的误差（如下图所示），且该误差从 7 为以后开始，考虑到 float 的精度本身就为 7 位左右，认为这个误差是可接受的，该方法是可行的，所以在程序中采用。

```
Float型求和结果: 1130722560.0000000000000000
Double型求和结果: 1130722617.726932764053345

C:\Users\张芷溪\Desktop\最终版(2)\single test\x64\Debug
\singletest.exe (进程 29516)已退出, 代码为 0 (0x0)。
按任意键关闭此窗口...
```

3.1.2 max 函数

该函数直接在循环中，通过逐一比较数组元素与当前最大值，将较大的值保存下来。

```

float mysum(const float data[], const int len) {
    double sum = 0.0; // 中间变量，用于累加总和
    for (size_t i = 0; i < len; i++) {
        // sum += log(sqrt(data[i]));
        sum += data[i];
    }
    return sum;
}

```

3.1.3 sort 函数

关于排序函数的选择，我们先后考虑了归并、快排、插排、桶排、堆排等。

综合考虑认为，快速排序在有序数组上可能退化，归并排序占用较大额外空间（不在 VS 里手动设置堆栈预留空间就会爆栈），桶排序对数据分布依赖性强，插入排序不适合大规模随机数据，且对于初始化有序的数组完全线性，看不出加速比，没什么意义。

而堆排序在时间复杂度上始终保持 $O(n\log n)$ ，空间复杂度仅为 $O(1)$ ，不占用额外内存，且对有序和随机数据都表现稳定，最终被选为我们的排序算法。主要涵盖堆化和排序两部分。

```

/* ... */
void heapify(float* arr, int idx, int heap_size) {
    int largest = idx; // 假设当前节点为最大节点
    int left_child = 2 * idx + 1; // 左子节点索引
    int right_child = 2 * idx + 2; // 右子节点索引

    // 比较左子节点
    if (left_child < heap_size && arr[left_child] > arr[largest])
        largest = left_child;

    // 比较右子节点
    if (right_child < heap_size && arr[right_child] > arr[largest])
        largest = right_child;

    // 如果最大值不是当前节点，交换并递归调整
    if (largest != idx) {
        mySwap(arr[idx], arr[largest]);
        heapify(arr, largest, heap_size);
    }
}

/* ... */
void heapSort(float* arr, int len) {
    // 构建最大堆
    for (int i = len / 2 - 1; i >= 0; i--)
        heapify(arr, i, len);

    // 逐步将最大元素移到末尾，并调整堆
    for (int i = len - 1; i >= 0; i--) {
        mySwap(arr[0], arr[i]);
        heapify(arr, 0, i);
    }
}

```

3.1.4 不加速单机输出结果（计算机一）

```
总时间:83.2142  
总和:1.13072e+09  
最大值:9.33377  
求和时间:0.250974  
最大值时间:0.108467  
排序时间:82.8548  
检查排序: 1
```

3.2. 加速版

结合 OpenMP 的多线程并行计算 AVX 指令集进行加速处理。

3.2.1 sum 函数

(1) 并行化:

显示指定线程数为 64，在 OpenMP 的并行区域内，每个线程创建一个局部 AVX 累加器 `local_sum`，并根据线程 ID 计算自己负责的数据块范围。

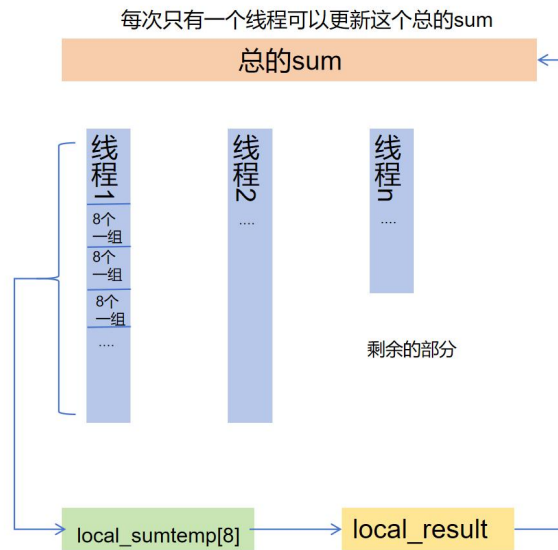
(2) AVX 加速:

每个线程通过 AVX 指令 `_mm256_load_ps` 一次性加载 8 个浮点数到 AVX 寄存器中，使用 `_mm256_add_ps` 进行向量加法（一次加 8 个浮点数），将结果累加到局部 AVX 寄存器中，得到 `local_sum`（这个数据也是 mm256 型，包含 8 个浮点数）。

完成 AVX 部分后，每个线程将这个 `local_sum` 的八个浮点数存储到一个对齐的临时数组中，累加为一个标量结果 `local_result`。如果数据量不是 8 的整数倍，则使用一个额外的 for 循环，将剩余的标量数据加到 `local_result` 中。即每个线程有自己的 `local_result`。

最后，通过 OpenMP 的 `#pragma omp critical` 指令，每个线程安全地将自己的局部和 `local_result` 加到全局的 `sum` 中，避免数据竞争。

最终，在所有线程完成任务后，返回存储在 `sum` 中的总和。



3.2.2 max 函数

(1) 并行化:

首先，函数通过 OpenMP 的 `#pragma omp parallel` 指令来并行化计算，`THREADS_NUM` 决定了使用多少个线程。每个线程负责处理输入数据的一部分。每个线程根据其 ID 和数据的长度 `len` 计算出自己负责处理的数据范围（`start` 到 `end`），并且通过 `chunk_size` 来确保每个线程大致处理相同数量的数据。最后一个线程负责处理剩余的元素。

(2) AVX 加速:

在每个线程内，使用 AVX 256 位寄存器（即 `__m256` 类型）来加速计算。首先，`max_vec` 被初始化为一个包含 `-FLT_MAX` 的向量，这保证了在后续计算中任何输入数据都比它大，从而不会影响最大值的更新。类型原因，`max_vec` 向量实际上存储的是当前线程处理的最大的 8 个值

接着，数据以 8 个浮点数为组进行处理（因为 AVX 寄存器一次可以存储 8 个浮点数）。每次循环通过 `_mm256_loadu_ps` 指令将数据加载到 `max_vec` 寄存器中，使用 `_mm256_max_ps` 指令将每组 8 个浮点数与 `max_vec` 中的当前最大值进行比较，并将比较后的结果更新回 `max_vec` 中。

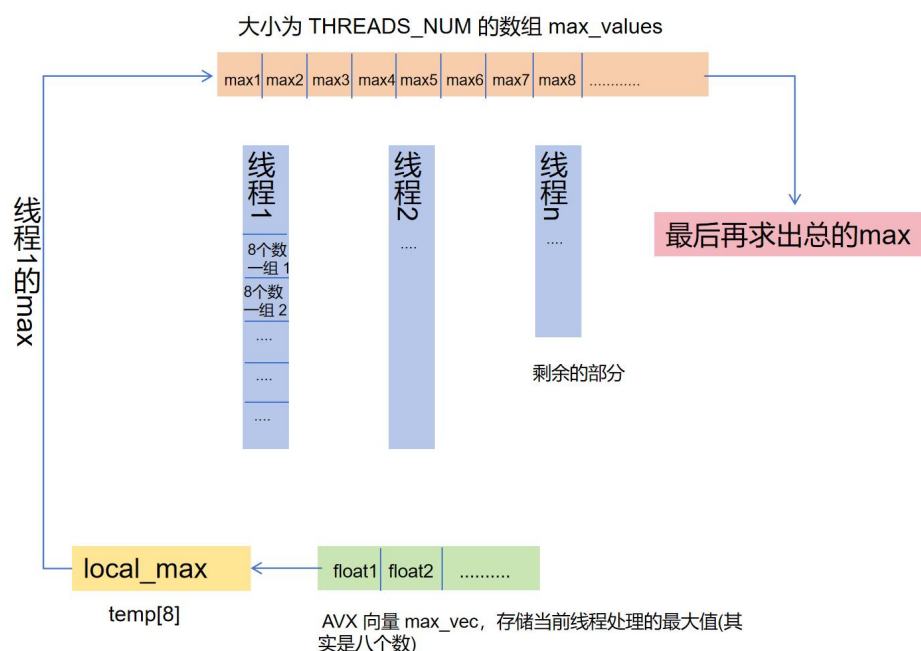
(3) 局部结果处理:

当一个线程内的所有数据通过 AVX 完成并行最大值计算后，`max_vec` 寄存器中存储了 8 个浮点数的最大值。为了进一步处理这些值，需要将 `max_vec` 中的 8 个浮点数提取到一个临时数组 `temp[8]` 中。此时，`temp` 数组的元素是 `max_vec` 中存储的 8 个最大值。

接下来，线程会在 `temp[8]` 数组中找出其中的最大值，这个值即为该线程处理的数据块的最大值（`local_max`）。对于数据块中无法整除 8 的剩余部分，线程会逐一检查这些元素，更新 `local_max`，确保它能包含所有数据的最大值。

(4) 汇总线程结果：

每个线程计算出的局部最大值 `local_max` 会被保存在 `max_values` 数组中，最后通过主线程汇总所有线程的结果。主线程将所有线程的 `local_max` 进行比较，得出最终的最大值 `global_max`，并返回该值作为函数的结果。



3.2.3 sort 函数

我们采用的堆排序并不适合进行 OpenMP 和 AVX 加速，主要原因在于其核心操作——堆的构建和堆调整（Heapify）存在显著的数据依赖性。

在堆的构建过程中，每个父节点的值都依赖于其子节点的状态，因此无法将堆结构轻易地拆分成多个独立、无依赖的子任务来交给不同线程并行处理。

而在堆排序的调整阶段，堆顶元素的提取和重新调整需要进行多次下沉操作（Heapify），这个过程是严格顺序依赖的，后续的调整必须依赖前一步调整的结果，导致线程之间难以独立操作。

此外，堆排序使用的是树形结构，数据存储在非连续的内存地址上，这与 AVX 指令集向量化处理的需求（即连续的内存块进行批量计算）不匹配，使得 AVX 很难高效地对堆结构进行 SIMD（单指令多数据）加速。尽管在堆构建的初始阶段可以尝试将堆分割成多个子树，由不同线程分别进行局部堆化处理，但

这种并行优化效果有限，且增加了线程同步的开销。

同时我们运行程序发现，仅对前两个函数进行处理，排序算法的结果也会有将近 10 倍的加速比：

```
Time Consumed: 8.87132s
Sum: 1.13066e+09
Max: 9.33377
Sum Time Consumed: 0.0159115s
Max Time Consumed: 0.0140991s
Sort Time Consumed: 8.84131s
检查排序：1
```

分析后认为，可能存在以下几方面原因：

1. CPU 缓存效应

（1）更好的缓存利用率

Sum 和 Max 加速的本质：向量化和优化内存访问减少了内存带宽占用，同时提高了缓存的利用率。

当 Sum 和 Max 的计算速度加快时，数据在 CPU 缓存（L1, L2, L3）中被更高效地访问和保留。

Sort 函数在运行时可能直接受益于已经在缓存中的数据，减少了缓存未命中（Cache Miss）。

（2）数据局部性改善

Sum 和 Max 的加速可能带来了更好的数据局部性（例如将数据更有序地预取到缓存中）。

Sort 函数可以直接利用这些已经优化的数据访问路径，减少不必要的内存等待时间。

2. CPU 资源争用减少

（1）减少了资源瓶颈

在未优化的情况下，Sum 和 Max 的计算可能占用大量的 CPU 端口和内存带宽，导致 Sort 在运行时受到资源争用的影响。

优化后的 Sum 和 Max 更高效地使用了 CPU 资源，释放出了更多的计算和内存带宽给 Sort。

（2）减少了缓存污染

Sum 和 Max 未被优化时，可能频繁导致缓存污染，驱逐掉 Sort 需要的数据。加速后的 Sum 和 Max 更高效地利用缓存，减少了缓存的污染，Sort 在执行时可以更稳定地访问到需要的数据。

3. 内存带宽优化

(1) 减少内存访问开销

优化后的 Sum 和 Max 减少了内存带宽的压力，使得内存控制器在 Sort 运行时可以更高效地响应内存请求。内存访问瓶颈的减少，使得 Sort 能够更快地读取和写回数据。就像一条单行道上有三辆车（Sum、Max 和 Sort）。Sum 和 Max 开得慢，导致 Sort 也被堵住了。现在 Sum 和 Max 开得快了，Sort 也可以流畅地开过去。

4. 指令缓存效应

(1) 更好的指令调度

在 Sum 和 Max 优化之后，CPU 内部的指令流水线和分支预测的效率可能提高，这也可能间接优化了 Sort 的执行环境，使其运行更加流畅。

5. 温度与频率效应

(1) 动态频率调整

如果 Sum 和 Max 在未优化的情况下长时间占用 CPU，优化后的 Sum 和 Max 减少了 CPU 的工作负载，使 CPU 可以在更高的频率下运行。Sort 在此时运行时，可能在更高的 CPU 频率下进行计算，从而表现得更快。

可能的因果链条？



四、双机版

4.1 运行环境

两台计算机使用手机热点连接在同一局域网下。

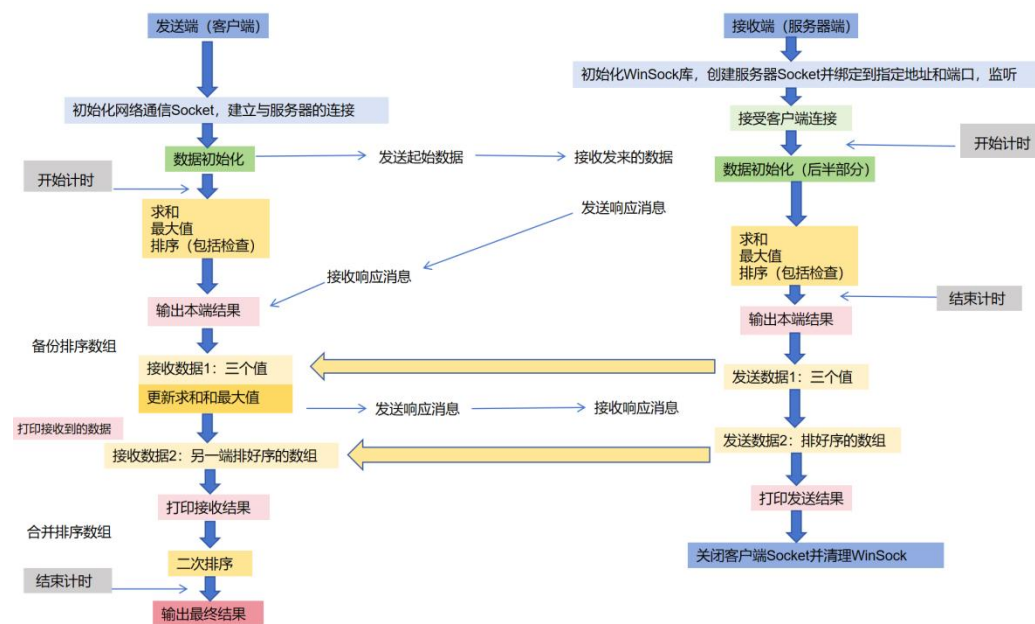
4.2 程序架构思路

双机版本（加速版）整体思路：

两台计算机分别初始化 640 万数据，并建立通讯连接（TCP）

1. 求和（多线程+AVX）
2. 求最大值（多线程+AVX）
3. 排序（分别采用堆排序）
4. 回传数据（求和值|最大值|排序后的 640 万数据）
5. 对两台计算机的计算结果进行综合（合并数据之后采用归并排序）得到最终的结果
6. 检查排序

4.3 完整建构设计：



如图所示，本项目的整体架构可分为准备、计算和处理三个部分。

1. 在准备阶段，两台计算机各自初始化 640 万个单精度浮点数，并分别对 socket 进行初始化。然后，由客户端向服务器发起请求，服务器接受客户端请求后，分配新的 socket 套接字，完成连接并等待通信。

2. 在计算阶段，两台计算机分别顺序调用了 `sumOpenMp_AVX`，`maxOpenMp_AVX_heapSort_AVX` 三个函数。其中，`sumOpenMp_AVX` 和 `maxOpenMp_AVX` 中分别创建了 64 个线程用于求和以及求最大值。

heapSort_AVX 函数则通过堆排序将数组转化为一个最大堆，并不断提取最大值，直到数组排序完成。此外，计算阶段的所有代码均使用 AVX 指令集加速。

3. 在结束处理阶段，服务器将计算结果传回客户端。此时，根据两台计算机各自的计算结果，完成最终的求和、求最大值，并再次调用归并函数将两个排序结果进行归并。然后，检查排序结果，输出排序结果是否正确，求和结果以及数据最大值。

4.4 关键代码及其分析

通讯重要部分分为两块：

1. 如何进行 tcp“三次握手”建立连接。
2. 如何传输最大值和排序结果给客户端。

4.4.1 tcp 面向连接“三次握手”

- 客户端首先通过 connect()发起连接请求（SYN）。
- 服务器通过 accept()响应请求并发送 SYN-ACK。
- 客户端发送 ACK 确认连接，三次握手完成，连接建立

(1) 客户端（Client）：

1. 初始化 WinSock:

客户端首先通过 WSASStartup 函数初始化 WinSock（Windows Socket），这是 Windows 操作系统中网络通信的基本库。

```
WSADATA wsaData;  
WORD version = MAKEWORD(2, 2);  
int wsResult = WSAStartup(version, &wsaData);
```

2. 创建 Socket:

客户端使用 socket()函数创建一个 TCP 套接字（SOCK_STREAM 表示 TCP 连接）。

```
SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0);
```

3. 发起连接请求:

客户端通过 connect（）函数向服务器发起连接请求，这就是三次握手的第一步。客户端向服务器发送一个带有 SYN 标志的 TCP 包，请求建立连接。

```
connect(clientSocket, (sockaddr*)&hint, sizeof(hint));
```

4. 等待服务器响应:

客户端会等待服务器的 SYN-ACK 响应。如果服务器接受连接，它将发送一个带有 SYN 和 ACK 标志的 TCP 包，表示连接请求被接受。

(2) 服务器端 (Server) :

服务器端会在 bind()和 listen()函数后进行监听，等待客户端的连接请求:

1. 初始化 WinSock:

和客户端一样，服务器首先需要初始化 WinSock 库。

```
WSADATA wsaData;  
WORD version = MAKEWORD(2, 2);  
int wsResult = WSStartup(version, &wsaData);
```

2. 创建 Socket:

服务器创建一个 TCP 套接字并绑定到指定端口。

```
SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0);  
  
sockaddr_in hint;  
hint.sin_family = AF_INET;  
hint.sin_port = htons(8899);  
hint.sin_addr.S_un.S_addr = INADDR_ANY;  
bind(serverSocket, (sockaddr*)&hint, sizeof(hint));
```

3. 监听并接受连接:

服务器使用 listen()函数开始监听指定端口上的客户端连接请求。当一个连接请求到达时，accept()函数会接受连接，并为客户端创建一个新的套接字 (clientSocket)。

```
listen(serverSocket, 5);  
sockaddr_in client;  
int clientSize = sizeof(client);  
SOCKET clientSocket = accept(serverSocket, (sockaddr*)&client, &clientSize);
```

这里的 accept()函数是 TCP 三次握手的第二步:

在客户端发送 SYN 请求后，服务器通过 accept()函数接收到 SYN 请求，并且服务器通过发送一个 SYN-ACK 包来响应客户端的请求，告诉客户端“我准备好了，连接建立”。服务器处于 SYN_RCVD 状态，准备与客户端继续通信。

4. 连接建立后交换数据:

当客户端收到服务器的 SYN-ACK 响应后，会发送一个 ACK 包确认连接建立。至此，TCP 连接已经建立，客户端与服务器可以开始交换数据。

```
char buffer[4096];
int bytesReceived;
while (true) {
    memset(buffer, 0, 4096);
    bytesReceived = recv(clientSocket, buffer, 4096, 0);
    if (bytesReceived == 0) {
        std::cerr << "Connection closed by Client" << std::endl;
        break;
    } else if (bytesReceived == SOCKET_ERROR) {
        std::cerr << "Can't receive data, Err #" << WSAGetLastError() << std::endl;
        break;
    } else if (bytesReceived < 4096) {
        std::cout << "Received: " << buffer << std::endl;
        send(clientSocket, "Received Data!", 14, 0);
    } else {
        std::cout << "Received: " << buffer << std::endl;
    }
}
closesocket(clientSocket);
WSACleanup();
```

服务器通过 `recv()` 接收客户端发送的消息。

4.4.2 传输最大值|和|排序结果。

数据传输通过 TCP 套接字实现。

(1) 服务器端 (Server) :

1. 发送结果数据给客户端 (SumResult、MaxResult 和 checkflag) :

在服务器端，首先将三个结果（总和、最大值、排序判断结果）封装成一个字符串，使用“|”符号作为分隔符，并使用 `send()` 函数将封装好的字符串发送到客户端。数据通过 `clientSocket` 连接发送。

```
// 将三个结果封装为字符串并发送给客户端
string dataString = std::to_string(SumResult) + "|" + std::to_string(MaxResult) + "|" + std::to_string(checkflag);
send(clientSocket, dataString.c_str(), dataString.size(), 0); // 发送处理结果给客户端
```

接收客户端反馈数据:

接收数据: `recv()` 函数从客户端接收数据并存储到缓冲区 `buffer` 中。如果 `bytesReceived == 0`，表示客户端关闭了连接。如果 `bytesReceived == SOCKET_ERROR`，则表示接收数据出错，`WSAGetLastError()` 提供错误码。


```

// 接收客户端的反馈数据
memset(buffer, 0, sizeof buffer); // 清空缓冲区

bytesReceived = recv(clientSocket, buffer, 4096, 0); // 接收数据
if (bytesReceived == 0) { // 如果接收到的数据长度为0, 表示客户端关闭了连接
    std::cerr << "Connection closed by server" << std::endl;
}
else if (bytesReceived == SOCKET_ERROR) { // 如果接收数据出错
    std::cerr << "Can't receive data, Err #" << WSAGetLastError() << std::endl;
}
std::cout << "Received: " << buffer << std::endl;

```

发送浮点数据（rawFloatData）给客户端：

‘rawFloatData’ 是浮点数数组，首先通过 ‘memcpy()’ 转换为字符数组 ‘rawCharArray’，每个浮点数占 4 字节。每次最多发送 4096 字节数据，确保不超过 TCP 数据包的最大负载。‘sentLength’ 用于追踪已发送的数据长度，确保每次发送的数据量不超过 4096 字节。通过循环发送，将所有数据分批发送，直到所有数据都发送完成。

```

// 将浮点数据数组转换为字符数组进行发送
memcpy(rawCharArray, rawFloatData, DATANUM * 4); // 将浮点数数组转换为字符数组

// 发送rawFloatData数据
int sentLength = 0; // 已发送的字节数
int charLength = DATANUM * 4; // float 类型占 4 字节, 算数据总字节数
while (sentLength < charLength) { // 循环发送数据
    int toSend = charLength - sentLength; // 计算还需发送的字节数
    if (toSend > 4096) { // 每次发送最大4096字节
        toSend = 4096;
    }
    for (int i = 0; i < toSend; i++) { // 将数据拷贝到缓冲区
        buffer[i] = rawCharArray[sentLength + i];
    }
    int bytesSent = send(clientSocket, buffer, toSend, 0); // 发送数据
    if (bytesSent == SOCKET_ERROR) { // 如果发送失败, 输出错误信息
        std::cerr << "Send failed" << std::endl;
        break;
    }
    sentLength += bytesSent; // 更新已发送字节数
}

```

发送完成后关闭连接并清理：

服务器端关闭与客户端的套接字连接，并清理 WinSock 库。

```

// 关闭客户端Socket并清理WinSock
closesocket(clientSocket); // 关闭Socket连接
WSACleanup(); // 清理WinSock库

```

(2) 客户端（Client）：

接收处理结果数据：

客户端通过 `Stringsplit(info, '|')` 函数解析接收到的数据字符串，提取 `SumResult`、`MaxResult` 和 `checkflag`。如果 `bytesReceived == 0`，表示服务器关闭了连接。如果 `bytesReceived == SOCKET_ERROR`，则表示接收数据出错。

```
// 接收网络数据
char buffer[4096]; // 定义缓冲区
int bytesReceived; // 接收的字节数
memset(buffer, 0, sizeof buffer); // 清空缓冲区，确保缓冲区是空的，防止脏数据影响后续解析
bytesReceived = recv(clientSocket, buffer, sizeof(buffer), 0); // 接收数据：从指定的 clientSocket 中读取数据到 buffer

if (bytesReceived == 0) {
    std::cerr << "Connection closed by server" << std::endl; // 服务器关闭连接
}
else if (bytesReceived == SOCKET_ERROR) {
    std::cerr << "Can't receive data, Err #" << WSAGetLastError() << std::endl; // 接收数据错误
}
```

3. 接收排序结果数据（浮点数组）：

```
//接下来，接收对方排好序的数组
int receivedLength = 0;
do {
    memset(buffer, 0, sizeof buffer); // 清空缓冲区
    bytesReceived = recv(clientSocket, buffer, sizeof(buffer), 0); // 接收数据
    for (size_t i = 0; i < bytesReceived; i++) {
        recieveCharData[i + receivedLength] = buffer[i]; // 将接收到的数据存储到recieveCharData,
    }
    receivedLength += bytesReceived; // 更新已接收字节数
} while (bytesReceived == 4096); // 如果缓冲区满，说明可能还有数据没收到，继续接收

printf("收到字节长度: %d , ", receivedLength);
```

分批接收数据：

由于浮点数组 `rawFloatData` 可能很大，无法一次性接收所有数据，因此客户端通过一个 `do...while` 循环分批接收数据。每次接收到的数据量最多为 4096 字节。

确保完整接收数据：

- `recv()` 函数每次会接收一个数据块（最多 4096 字节），如果缓冲区没有接收到完整数据（即 `bytesReceived == 4096`），客户端会继续接收，直到所有数据都被接收完。

- 每次接收到的数据都会追加到 `recieveCharData` 数组中，确保数据按顺序接收。

- 并且客户端最终输出接收到的数据字节长度，确认接收的数据量。

4.5 如何保证数据完整性和不丢包：

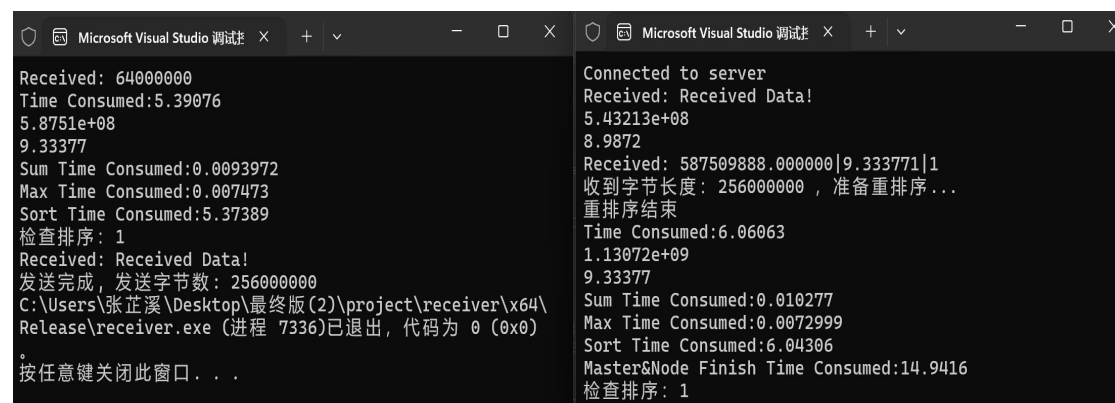
1. TCP 协议的可靠性：TCP 本身是面向连接的、可靠的协议，保证数据包的顺序和完整性。在传输过程中，TCP 会自动处理丢包、重传和校验等问题，确保数据不丢失。

2. 分批发送/接收：服务器端和客户端都通过分批发送/接收的方式处理较大的数据（例如浮点数数组）。每次发送或接收数据的大小都受到缓冲区的限制（最大 4096 字节）。这种方式可以确保数据按顺序传输，并且不丢失。

3. 接收数据的累积存储：客户端通过 `receiveCharData` 数组累积接收到的数据，确保每次接收的数据能够被拼接成完整的数据。`receivedLength` 变量用于追踪已接收的字节数，避免覆盖之前接收到的数据。

分割字符串数据：对于简单的字符串数据（如总和、最大值、排序判断结果），客户端使用分隔符 `|` 将数据分割成多个部分，确保每个数据部分都可以被正确解析。

4.6 输出结果展示



```
Received: 64000000
Time Consumed:5.39076
5.8751e+08
9.33377
Sum Time Consumed:0.0093972
Max Time Consumed:0.007473
Sort Time Consumed:5.37389
检查排序: 1
Received: Received Data!
发送完成, 发送字节数: 256000000
C:\Users\张芷溪\Desktop\最终版(2)\project\receiver\x64\
Release\receiver.exe (进程 7336)已退出, 代码为 0 (0x0)
。
按任意键关闭此窗口...

Connected to server
Received: Received Data!
5.43213e+08
8.9872
Received: 587509888.000000|9.333771|1
收到字节长度: 256000000, 准备重排序...
重排序结束
Time Consumed:6.06063
1.13072e+09
9.33377
Sum Time Consumed:0.010277
Max Time Consumed:0.0072999
Sort Time Consumed:6.04306
Master&Node Finish Time Consumed:14.9416
检查排序: 1
```

五、结果对比分析

总共列出单机不加速、单机加速、双机通讯不重排序、双机通讯重排序的数据。

其中，“双机通讯不重排序”是指，考虑到本项目双机初始化传输时顺序初始化，且显式传输规定了一端初始化前半，一端初始化后半，发送数组时直接合并并就能得到顺序数组，故接收到顺序数组后直接进行检查排序，发现排序正

确，省略掉重排序过程，因为是并行运算，故直接取单机最长时间作为总时间，但该方法只适用于一端数据最大值确定小于等于另一端最小值的情况，所以也给出完全重排序的结果。

以下是详细数据：

单位：秒

5.1 单机不加速版（单机数据量 64*2000000）：

Sum 函数

次数	计算机一	计算机二
一	0.294379	0.230382
二	0.251856	0.232737
三	0.252262	0.234439
四	0.261059	0.242134
五	0.264126	0.22923
平均值	0.2647364	0.2337844
最终平均	0.2492604	

max 函数

次数	计算机一	计算机二
一	0.138371	0.0952601
二	0.100248	0.100079
三	0.108804	0.0978713
四	0.109966	0.100868
五	0.126252	0.10477
平均值	0.1167282	0.09976968
最终平均	0.10824894	

sort 函数

次数	计算机一	计算机二
一	84.8437	85.3999
二	84.5401	84.6998
三	83.4263	85.9674
四	83.5224	85.6145
五	83.5962	82.945
平均值	83.98574	84.92532
最终平均	84.45553	

总时间

次数	计算机一	计算机二
一	85.2765	85.7255
二	84.8923	85.0326
三	83.7874	86.2997
四	83.8934	85.9575
五	83.9866	83.279
平均值	84.36724	85.25886
最终平均	84.81305	

5.2 单机加速版（单机数据量 64*2000000）：

Sum 函数

次数	计算机一	计算机二
一	0.0165415	0.0226813
二	0.0164977	0.0229343
三	0.0156741	0.0197429
四	0.0158349	0.0193241
五	0.015663	0.0505907
平均值	0.01604224	0.02705466
最终平均	0.02154845	

max 函数

次数	计算机一	计算机二
一	0.0146224	0.0107033
二	0.0140699	0.0106709
三	0.0139328	0.0102906
四	0.0139824	0.0110817
五	0.0139123	0.0142597
平均值	0.01410396	0.01140124
最终平均	0.0127526	

sort 函数

次数	计算机一	计算机二
一	9.02764	8.84655
二	8.60596	8.64283
三	9.00047	8.60586
四	8.57989	8.79563
五	8.67121	8.66081

平均值	8.777034	8.710336
最终平均	8.743685	

总时间

次数	计算机一	计算机二
一	9.05881	8.87994
二	8.63653	8.67643s
三	9.03007	8.6359
四	8.60971	8.82604
五	8.70079	8.72567
平均值	8.807182	8.7668875
最终平均	8.78703475	

5.3 双机加速版不重排序（单机数据量 64*1000000）：

max 函数

次数	计算机一	计算机二
一	0.0074665	0.0045594
二	0.0073045	0.0045310
三	0.0076987	0.0044530
四	0.0080016	0.0043938
五	0.0076374	0.0047562
平均值	0.0061968	

Sum 函数

次数	计算机一	计算机二
一	0.0091424	0.0172071
二	0.0101544	0.0243565
三	0.0123183	0.0147816
四	0.0101653	0.0267586
五	0.0090206	0.0164419
平均值	0.01273125	

sort 函数

次数	计算机一	计算机二
一	4.32038	4.79496
二	4.22651	4.68737
三	4.38763	4.57696
四	4.39602	4.48935

五	4.79755	4.59521
平均值	4.69638	

总时间

次数	计算机一	计算机二
一	4.33699	4.81672
二	4.24397	4.71625
三	4.40764	4.59619
四	4.41418	4.52051
五	4.8142	4.61641
平均值	4.715305	

5.4 双机加速版重排序（单机数据量 64*1000000）：

sort 函数

次数	计算机一	计算机二
一	4.83184	4.27073
二	4.84912	4.32371
三	4.7092	4.87404
四	4.84356	4.32541
五	4.89651	4.3391
平均	4.826046	4.426598
最终平均	4.626322	

总时间

次数	计算机一
一	11.9084
二	11.8164
三	12.5949
四	12.0696
五	12.0693
平均	12.09172

5.5 加速比统计：

类别	Max	Sum	Sort	总时间	Sum	Max	Sort	总加速比
单机不加速版	0.10824894	0.2492604	84.45553	84.81305	1	1	1	1

单机加速版	0.0127526	0.02154845	8.743685	8.78703475	9	12	10	16
双机加速不重排序	0.0061968	0.01273125	4.69638	4.715305	18	20	18	29
双机加速重排序			4.998434	12.81384			17	11

单机加速版：相较于单机不加速版，加速版在整体性能上有了显著提升；
双机加速充分发挥了双机并行计算的优势，实现了更高的加速效果，大幅缩短了计算时间；
但由于重排序操作的影响，重排序排序对性能产生了较大的负面影响，使总加速比相对较低。

六、总结和心得

6.1 收获

- 1. 技术应用：通过本次项目，我们掌握了 OpenMP 多线程并行计算和 AVX 指令集的使用方法，以及 TCP 通讯技术在分布式计算中的应用。
- 2. 性能优化：通过本次项目，对加速优化有了一定的了解，体验了整个流程，也对内存、指令集、多线程的认识更加深入。
- 3. 数据传输与通讯：项目中涉及了数据在两台计算机之间的传输和通讯，我们学会了如何使用 TCP 协议建立可靠的连接，并确保数据的完整性和准确性，相关思维也有了提升。

6.2 仍可改进的方向

- 1. 重排序优化：尽管双机加速版在不重排序的情况下表现优异，但在需要重排序的场景下，性能受到了较大影响。可以探索更高效的重排序算法，减少数据传输和排序的开销，提高重排序的效率。
- 2. 动态负载均衡：目前的实现中，线程和计算任务的分配是静态的，未来可以引入动态负载均衡机制，根据各线程和计算机的实时负载情况，动态调整任务分配，进一步提高资源利用率和计算效率。
- 3. 错误处理与容错机制：在分布式计算中，网络故障、计算机故障等不可预见的问题可能导致计算任务失败。未来可以引入更完善的错误处理和容错机

制，确保系统在出现故障时能够自动恢复，继续完成计算任务。

4. 性能监控与调优：在实际应用中，性能监控和调优是持续优化系统的关键。未来可以开发性能监控工具，实时监控系统运行状态，及时发现性能瓶颈，并进行针对性的调优。

总而言之，通过本次项目，我们不仅实现了既定目标，还积累了宝贵的经验和教训。这些收获和改进方向将为我们未来在高性能计算和分布式系统领域的研究和开发提供重要的参考。

七、分工

单机部分、报告一、二部分：张芷溪

双机部分、报告三、四部分：曹雨琦

报告五、六部分一起完成，实际代码部分不完全分割，互有交叉。