算法大作业报告

课　　程　算法

学　　院　电子与信息工程学院

组　　员　崔屿杰　　学号　2151105

任课教师　王晓年

时　　间　2024 年 1 月 14 日

目录

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：最后加速比以我测试为准。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，sd::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，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);
}
```

二、程序架构设计

本大作业程序实现了 Linux 系统上的 Server 端与 Windows 系统上的 Client 端之间的通信。两机先各自生成数据量为 64*1000000 的数组，并进行打乱顺序。首先不加速计算求和，client 发送结果到 server 端，进行合并；不加速求最大值，client 端发送结果到 server 端，进行求结果最大值；两机各不加速用普通归并排序排序一半数组，client 传输结果到 server 端，server 端进行两数组合并起来的排序。然后是加速计算求和、求最大值、排序，流程一样，就是运算的时候使用了加速方法，具体是使用了 OPENMP 加速和 sse 加速。

具体流程如图 1：

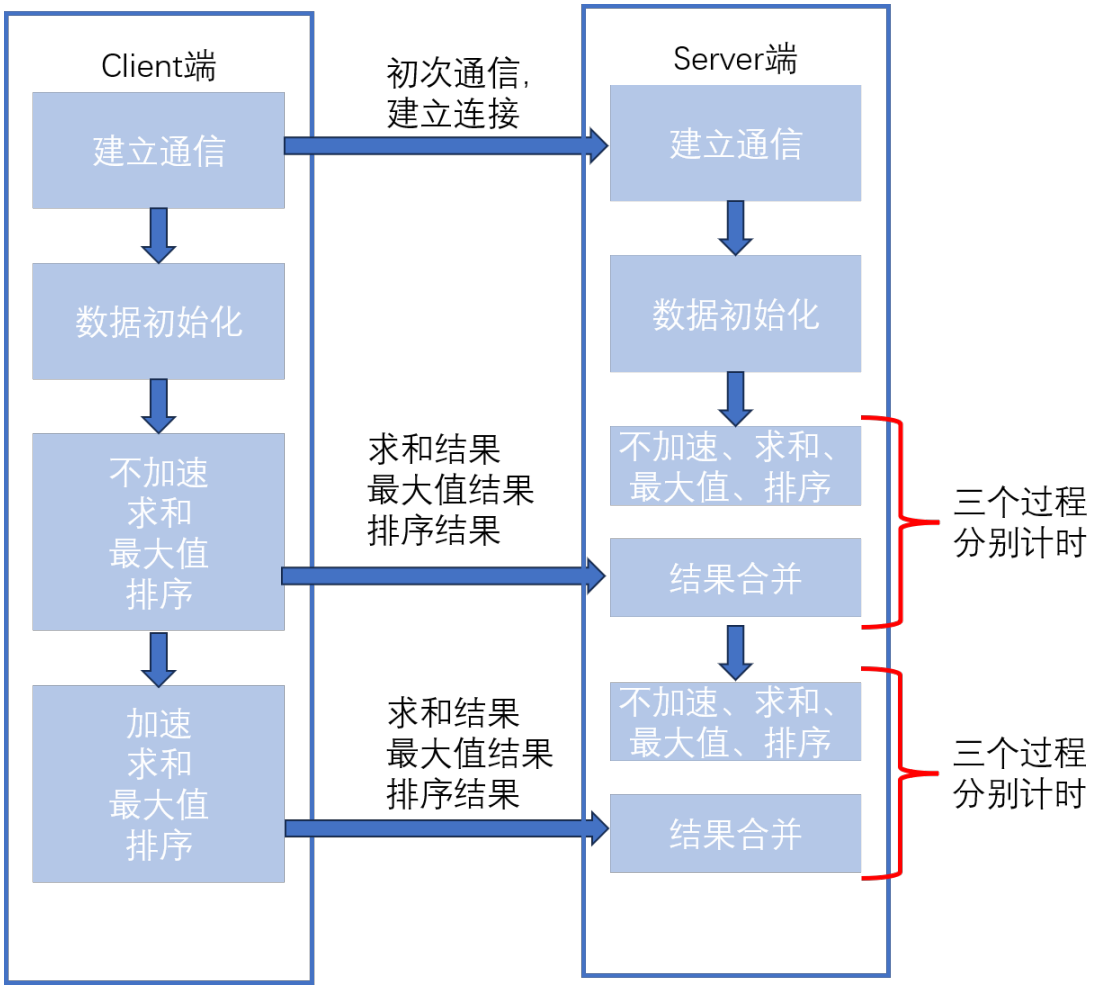

图 1

三、设计原理

3.1 多线程

多线程是一种允许单个程序同时运行多个操作的编程概念。线程是程序执行流的最小单位，它是进程的一个实体。一个进程可以包含多个线程，每个线程执行不同的任务。

多线程的原理是操作系统负责线程的创建、调度和管理。线程共享其所属进程的资源，如内存和文件句柄。当操作系统从一个线程切换到另一个线程时，它保存当前线程的状态并加载新线程的状态，这称为上下文切换。多线程可以是并行的（多个线程在多核处理器上同时执行）或并发的（在单核处理器上快速交替执行，给人同时执行的错觉）。本实验是采用并发执行机制，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 。

多线程允许程序同时执行多个操作，这可以显著提高应用程序的效率和响应速度。线程

共享其父进程的资源，这使得线程间的数据共享和通信更为高效。他的优点是在多核处理器上，多线程可以有效地利用 CPU 资源，加快任务执行速度。

然而多线程也存在一定的缺点：线程间共享数据可能导致数据不一致，需要使用同步机制，如锁，来避免冲突，这可能导致复杂性增加。线程虽比进程轻，但创建和管理线程仍然消耗资源。

3.1.1 OpenMP

本实验的加速方案中，多线程加速手段具体是采取了 OpenMP。OpenMP 是一个支持跨平台共享内存多处理编程的标准。它定义了一组编译器指令、库函数以及环境变量，旨在开发高效的并行应用程序。OpenMP 主要用于 C、C++和 Fortran 语言。

选用 OpenMP，是因为 OpenMP 简化了共享内存多线程编程，通过并行化计算密集型任务，OpenMP 能够显著提高应用程序的性能，尤其是在多核心和多处理器系统上。OpenMP 程序可以根据可用的处理器核心数自动扩展，从而提高程序的可扩展性。

此外，OpenMP 支持多种编译器和操作系统，使得并行代码可以在不同平台上运行。减小了本实验在 Linux 和 Windows 双系统上编程调试的成本。

OpenMP 使用指令（通常是以 #pragma omp 开头的预处理器指令）来标识并行区域。编译器根据这些指令生成多线程代码。它提供了一个运行时环境，它负责管理线程的创建、执行和终止。它允许指定数据是共享的还是每个线程私有的，以及如何将数据传递到并行区域。

OpenMP 采用 fork-join 的执行模式。开始的时候只存在一个主线程，当需要进行并行计算的时候，派生出若干个分支线程来执行并行任务。当并行代码执行完成之后，分支线程会合，并把控制流程交给单独的主线程。原理如图 2。

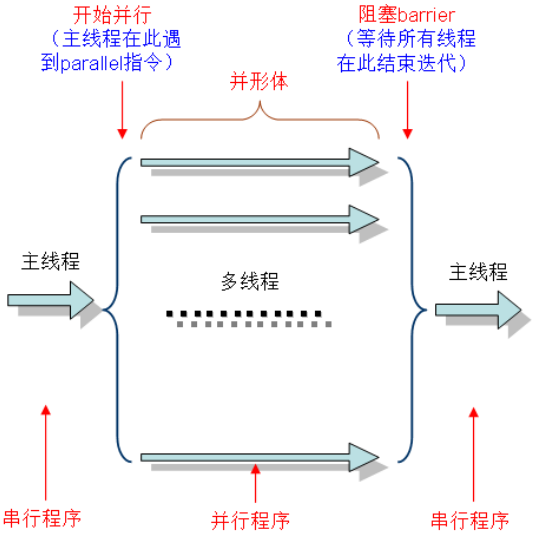

图 2

3.2 SSE 加速

SSE 是一种 SIMD（Single Instruction, Multiple Data）架构，它允许一条指令同时对多个

数据进行操作。它是一种由英特尔公司开发的 CPU 指令集扩展，用于加速处理器上的多媒体、通信和信号处理应用。

SSE 经历了多次迭代和改进。其主要目的是为了提高浮点运算的效率，特别是在 3D 图形领域。SSE2、SSE3 和后续版本的 SSE4 等进一步扩展了这些功能，增加了对整数和双精度浮点数的支持，以及更多针对特定类型计算的优化指令。

在 SSE 中，引入了特殊的 128 位寄存器，这些寄存器可以同时存储多个较小的数据值，例如四个 32 位浮点数。这样，一条指令就可以同时对这四个数进行相同的操作，比如加法或乘法，从而显著提高数据处理速度。这种并行处理方式在图形处理、视频编解码、音频处理和各类科学计算等领域特别有用，因为这些领域通常涉及到大量类似的运算。

使用 SSE 指令集的主要优势在于其能显著提高程序的性能，尤其是在浮点运算密集型任务中。但是也遇到了在部分处理器、操作系统不兼容的问题。例如在 linux 系统 ubuntu20.04 的最新版 GCC 与 Clang 中并不支持对于 log 运算的加速。

3.3 TCP 通信

TCP（Transmission Control Protocol）是一种广泛使用的网络通信协议，它位于互联网协议套件的传输层。TCP 的设计目标是提供一种可靠的、面向连接的通信方式。这意味着在数据开始传输之前，两个网络端点（通常是两台计算机）需要建立起一个连接，并在整个会话期间维护这个连接。

TCP 的工作原理是基于将数据分割成称为“数据包”的小块进行传输。每当发送一个数据包，发送方都会等待接收方发送回一个确认信号，表明数据包已经被成功接收。如果确认信号在一定时间内没有收到，发送方会重新发送该数据包。这种机制确保了即使在网络条件不佳的情况下，数据也能可靠地传输到目的地。

三次握手是 TCP 建立连接的一个关键过程。这个过程包括以下三个步骤：

（1）发起连接：首先，客户端发送一个 SYN（同步序列编号）数据包到服务器，以开始一个新的连接。这个动作实际上是客户端告诉服务器：“我想开始与你通信。”

（2）响应和确认：服务器接收到这个 SYN 数据包后，会回送一个 SYN-ACK（同步和确认）数据包。这个步骤是服务器对客户端的回应，表示：“我收到了你的请求，并准备好了建立连接。”

（3）客户端确认：最后，客户端再次发送一个 ACK（确认）数据包给服务器，以确认收到了服务器的 SYN-ACK 响应。这时，连接就正式建立了，双方可以开始数据传输。

通过这种方式，TCP 确保了两个网络端点在数据传输开始前能够协商并建立起一个稳定的连接。这种三次握手机制是 TCP 可靠性的关键所在，它确保了连接的两端都准备好了数据传输，并且能够应对网络延迟或其他可能的干扰。

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

```
1.  //不加速求和
2.  float sum(const float data[], const int len)
3.  {
4.      double result = 0.0;
5.      double c = 0.0;
6.      for (int i = 0; i < len; i++) {
7.          double corrected = log(sqrt(data[i])) - c;    // 当前值和之前累积的误差的差
8.          double newresult = result + corrected; // 将修正后的值加到总和中
9.          c = (newresult - result) - corrected;  // 计算并保存新的小误差
10.         result = newresult;
11.     }
12.
13.     return static_cast<float>(result);
14. }
```

该函数使用了 Kahan 求和算法，又名补偿求和或进位求和算法，是一个用来降低有限精度浮点数序列累加值误差的算法。它主要通过保持一个单独变量用来累积误差（常用变量名为 c）来完成的。

在计算机程序中，我们需要用有限位数对实数做近似表示，如今的大多数计算机都使用 IEEE-754 规定的浮点数来作为这个近似表示。对于如 1/3 由于我们不能在有限位数内对它进行精准表示，因此在使用 IEEE-754 表示法时，必须四舍五入一部分数值（truncate）。这种 舍入误差（Rounding off error）是浮点计算的一个特征。

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。

4.1.2 求最大值算法

```
1.     //不加速求最大值
2.  float max(const float data[], const int len)
3.  {
4.      double result = 0.0f;
5.      for (int i = 0; i < len; i++){
6.          if (result <= log(sqrt(data[i])))
```

```
            result = log(sqrt(data[i]));
        }
        return result;
}
```

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，则将最大值置换为该元素，遍历结束后即得到最大值。

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。

4.1.3 排序算法

```
//不加速排序、判断排序是否正确
void normal_sort(float* A, int x, int y, float* T)
{
    if (y - x > 1) {
        int m = x + (y - x) / 2;
        int p = x, q = m, i = x;
        normal_sort(A, x, m, T);
        normal_sort(A, m, y, T);
        while (p < m || q < y) {
            if (q >= y || (p < m && log(sqrt(A[p])) <= log(sqrt(A[q])
))) {
                T[i++] = A[p++];
            }
            else {
                T[i++] = A[q++];
            }
        }
        for (i = x; i < y; i++) {
            A[i] = T[i];
        }
    }
}

bool isSortTrue(float data[]) {
    for (size_t i = 0; i < sizeof(data) / sizeof(data[0]); i++)
        if (data[i] > data[i + 1])
            return false;
    return true;
}
```

排序采用的是归并排序，选用它是因为这是一种给稳定排序，不受数组有序程度的影响，

可以保持稳定的时间复杂度，利于排除数组打乱结果有序性程度的影响，更直观体现加速手段的有效性。

本函数的归并排序采用递归去实现，每个递归过程有三个步骤：

（1）分解：把待排序的 n 个元素的序列分解为两个子序列，每个子序列包括 n/2 个元素。

（2）治理：对每个子序列分别调用归并排序 normal_sort，进行递归操作。

（3）合并：合并两个排好序的子序列，生成排序结果。

最后，不管是单独排序还是合并排序的结果，都使用 isSortTrue 进行检验。

4.1.4 双机排序结果合并

```
1.  //双机排序结果合并
2.  void SortResultCombine(float* StrA, int lenA, float* StrB, int lenB,
    float* StrC)
3.  {
4.      int i, j, k;
5.      i = j = k = 0;
6.      while (i < lenA && j < lenB) {
7.          if (StrA[i] < StrB[j]) {
8.              StrC[k] = StrA[i];
9.              i++;
10.             k++;
11.         }
12.         else {
13.             StrC[k] = StrB[j];
14.             k++;
15.             j++;
16.         }
17.     }
18.     while (i < lenA) {
19.         StrC[k] = StrA[i];
20.         k++;
21.         i++;
22.     }
23.     while (j < lenB) {
24.         StrC[k] = StrB[j];
25.         k++;
26.         j++;
27.     }
28. }
```

这段函数实现了两个已排序数组的合并排序。它假设输入的两个数组 StrA 和 StrB 都已经是升序排序的，并将它们合并为一个新的升序数组 StrC。这是通过比较两个数组的元素，并按顺序将较小的元素先加入到结果数组 StrC 中实现的。

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

```
1.    // 主加速求和函数
2.  float sum_speedup(const float data[], int len) {
3.      double totalSum = 0.0;
4.      int numThreads = omp_get_max_threads();//获得最大线程数
5.      int chunkSize = len / numThreads;//分块，每个线程分工
6.
7.      // OpenMP 并行区域
8.      #pragma omp parallel reduction(+:totalSum)//规约操作，将每个线程的值累加到原始的 totalSum 变量中
9.      {
10.         int threadId = omp_get_thread_num();//线程 ID
11.         int start = threadId * chunkSize;
12.         int end = (threadId + 1) == numThreads ? len : (threadId + 1) * chunkSize;
13.
14.         __m256 vdata, vroot;
15.         float rootArray[8];
16.
17.         //每次循环处理 4 个浮点数，SSE 指令操作 4 个单精度浮点数
18.         for (int i = start; i < end; i += 8) {
19.             vdata = _mm256_loadu_ps(&data[i]); // 使用 SSE 加载 8 个浮点数到 256 位 sse 寄存器内，加载的数据不需要在内存中对齐
20.             vroot = _mm256_sqrt_ps(vdata); // 并行计算平方根
21.             vroot = _mm256_sqrt_ps(vroot); // 再进行一次平方根元算，代替 log 运算（linux 的 gcc 和 clang 里没有该函数）
22.             _mm256_storeu_ps(rootArray, vroot); // 将结果存储到数组中
23.
24.             for (int j = 0; j < 4; ++j) {
25.                 totalSum += rootArray[j];
26.             }
27.         }
28.     }
29.     return totalSum;
30. }
```

这个函数是一个加速版的求和函数，用于计算一个浮点数数组 data 的总和。它通过使用 OpenMP 和 SSE 指令集实现并行处理和向量化计算，从而加速运算。

（1）OpenMP 并行化：代码使用 OpenMP 来创建并行区域，允许多个处理器同时执行

代码。它首先获取可用的最大线程数，然后将数组分割成等大小的块，每个线程处理一个块。

（2）向量化计算：使用 SSE 指令集（在此代码中为 256 位宽的寄存器 __m256）进行向量化计算。向量化允许一次处理多个数据点，而不是单个数据点，从而显著提高性能。

（3）计算过程：每个线程分别计算其分配的数组块中的元素。它们首先使用 SSE 指令加载 8 个浮点数，计算这些数的平方根两次（作为对数运算的近似），然后将结果累加到总和中。

（4）规约操作：OpenMP 的规约操作 reduction(+:totalSum) 用于在所有线程间安全地累加各自计算的和到最终的总和 totalSum 中。

4.2.2 利用多线程加速的求最大值算法

```
1.  float max_speedup(const float data[], int len) {
2.      double maxResult = -INFINITY;
3.      int numThreads = omp_get_max_threads();
4.      int chunkSize = len / numThreads;
5.
6.      // OpenMP 并行区域
7.      #pragma omp parallel
8.      {
9.          int threadId = omp_get_thread_num();
10.         int start = threadId * chunkSize;
11.         int end = (threadId + 1) == numThreads ? len : (threadId + 1)
     * chunkSize;
12.
13.         double localMax = -INFINITY;
14.         __m256 vdata, vroot;
15.         float rootArray[8];
16.         double logVal;
17.
18.         for (int i = start; i < end; i += 8) {
19.             vdata = _mm256_loadu_ps(&data[i]); // 使用 SSE 加载 8 个浮
    点数
20.             vroot = _mm256_sqrt_ps(vdata); // 计算平方根
21.             vroot = _mm256_sqrt_ps(vroot); // 再进行一次平方根元算，代替
    log 运算（linux 的 gcc 和 clang 里没有该函数）
22.             _mm256_storeu_ps(rootArray, vroot); // 将结果存储到数组中
23.
24.             for (int j = 0; j < 4; ++j) {
25.                 logVal = rootArray[j];
26.                 if (logVal > localMax) {
27.                     localMax = logVal; // 更新局部最大值
28.                 }
29.             }
```

```
        }

        #pragma omp critical
        if (localMax > maxResult) {
            maxResult = localMax;
        }
    }
    return maxResult;
}
```

这个函数是一个加速版的求最大值函数，其目的是在一个浮点数数组 data 中找到最大值的平方根的平方根（作为对数运算的近似，Linux 上不支持 sse 对 log 的加速）。为了提高效率，它采用了 OpenMP 并行处理和 SSE 向量化指令集。

函数的核心思想是将数组分割成多个块，并利用多线程并行地在每个块中查找局部最大值。每个线程处理一部分数据，使用 SSE 指令一次处理 8 个元素，以提高计算效率。每个线程先计算其负责部分的最大值，然后将这些局部最大值合并以找到全局最大值。

（1）并行处理：使用 OpenMP 创建多线程，每个线程处理数组的一部分，以并行方式寻找各自部分的最大值。

（2）向量化计算：利用 SSE 指令集进行向量化计算，一次处理 8 个元素，提高处理速度。

（3）查找局部最大值：每个线程计算其分配块的最大值，使用 SSE 指令进行平方根计算，以近似对数运算。

（4）合并局部最大值：使用 OpenMP 的 critical 区域确保线程安全地更新全局最大值变量。

4.2.3 利用 OpenMP 加速的排序算法

```
float sqrtA[DATANUM];
void normal_sort_speedup(float* A, int x, int y, float* T)
{
    if (y - x > 1) {
        int m = x + (y - x) / 2;
        int p = x, q = m, i = x;
    //并行创建两个项目，分别处理数组两半
#pragma omp task shared(A, T, sqrtA)
        normal_sort_speedup(A, x, m, T);

#pragma omp task shared(A, T, sqrtA)
        normal_sort_speedup(A, m, y, T);

#pragma omp taskwait//等待之前创建的所有任务完成


        for (int i = x; i < y; i += 8) {
```

```
            __m256 vdata = _mm256_loadu_ps(&A[i]);
            __m256 vsqrt = _mm256_sqrt_ps(vdata);
            vsqrt = _mm256_sqrt_ps(vsqrt); // 再进行一次平方根元算，代替log运算（linux的gcc和clang里没有该函数）
            _mm256_storeu_ps(&sqrtA[i], vsqrt);
        }

        while (p < m || q < y) {
            if (q >= y || (p < m && A[p] < A[q])) {
                T[i++] = A[p++];
            }
            else {
                T[i++] = A[q++];
            }
        }
        for (i = x; i < y; i++) {
            A[i] = T[i];
        }
    }

}


bool isSortTrue_speedup(float data[]) {
    bool sorted = true;

#pragma omp parallel for
    for (size_t i = 0; i < sizeof(data)/sizeof(data[0]); i++) {
#pragma omp critical//保在任何给定时间只有一个线程可以执行
        if (data[i] > data[i + 1])
            sorted = false;

    }
    return sorted;
}
```

normal_sort_speedup 函数用于对浮点数数组 A 进行排序，并同时计算数组元素的平方根的平方根（作为对数运算的近似），存储在 sqrtA 数组中。它采用了并行处理和向量化计算来加速这个过程。

函数的主要思想是使用归并排序算法，并将其与 OpenMP 任务并行和 SSE 向量化相结合。归并排序是一种分治算法，通过递归地将数组分成两半进行排序，然后合并这两个已排序的半部分。

（1）并行递归排序：使用 OpenMP 任务 (#pragma omp task) 创建两个并行任务，分别递归地对数组的两个子部分进行排序。

（2）向量化计算：在每个递归调用结束后，使用 SSE 指令集处理数组的一部分，以向

量化方式计算元素的平方根的平方根，并将结果存储在 sqrtA 中。

（3）合并排序结果：使用归并排序的合并步骤将两个已排序的子数组合并成一个完整的排序数组。

isSortTrue_speedup 函数检查数组是否已经排序。它使用 OpenMP 并行循环来遍历数组，比较相邻元素以确认数组是否按升序排列。由于并行处理，每个线程检查数组的一部分，但通过使用 omp critical 来保证在修改 sorted 变量时的线程安全。

4.2.4 双机排序结果合并

```
1.  void SortResultCombine_speedup(float* StrA, int lenA, float* StrB, in
    t lenB, float* StrC) {
2.      // 并行区域
3.      #pragma omp parallel
4.      {
5.          // 获取线程数和当前线程 ID
6.          int numThreads = omp_get_num_threads();
7.          int threadId = omp_get_thread_num();
8.
9.          // 计算每个线程要处理的数据量
10.         int chunkSizeA = lenA / numThreads;
11.         int chunkSizeB = lenB / numThreads;
12.
13.         // 计算每个线程处理的数据段的起始和结束位置
14.         int startA = chunkSizeA * threadId;
15.         int endA = (threadId == numThreads - 1) ? lenA : startA + chu
    nkSizeA;
16.
17.         int startB = chunkSizeB * threadId;
18.         int endB = (threadId == numThreads - 1) ? lenB : startB + chu
    nkSizeB;
19.
20.         int startC = startA + startB;
21.         int i = startA, j = startB, k = startC;
22.
23.         // 合并两个数组的部分
24.         while (i < endA && j < endB) {
25.             if (StrA[i] < StrB[j]) {
26.                 StrC[k++] = StrA[i++];
27.             } else {
28.                 StrC[k++] = StrB[j++];
29.             }
30.         }
```

```
31.
32.         // 处理剩余的元素
33.         while (i < endA) {
34.             StrC[k++] = StrA[i++];
35.         }
36.         while (j < endB) {
37.             StrC[k++] = StrB[j++];
38.         }
39.     }
40. }
```

SortResultCombine_speedup 函数用于合并两个已排序的浮点数数组 StrA 和 StrB，将合并结果存储在数组 StrC 中。该函数采用了 OpenMP 并行化技术，允许多个线程同时处理不同部分的数据，以提高效率。

（1）并行化处理：使用 OpenMP 并行化机制，将工作分配给多个线程，每个线程负责处理一部分数据。线程数由 omp_get_num_threads() 获取。

（2）分配工作：根据线程数和数组长度，计算每个线程要处理的数据量，以及每个线程处理数据的起始和结束位置。

（3）合并数据：每个线程独立地合并其分配的部分。通过比较 StrA 和 StrB 中的元素，将较小的元素添加到结果数组 StrC 中，同时更新索引 i、j 和 k 来跟踪每个数组的当前位置。

（4）处理剩余元素：如果其中一个数组的元素已全部合并，而另一个数组还有剩余元素，那么剩余元素将被添加到结果数组 StrC 中。

4.3 网络通信

4.3.1 建立通信

Server 端：

```
1.     //TCP 通信
2.      int server_fd, new_socket;
3.      struct sockaddr_in address;
4.      int addrlen = sizeof(address);
5.
6.      server_fd = socket(AF_INET, SOCK_STREAM, 0);
7.      if (server_fd == 0) {
8.          perror("socket failed");
9.          exit(EXIT_FAILURE);
10.     }
11.
12.     address.sin_family = AF_INET;
13.     address.sin_addr.s_addr = INADDR_ANY;//接受所有
14.     address.sin_port = htons(8080);
```

```
15.
16.     //绑定套接字
17.     bind(server_fd, (struct sockaddr *)&address, sizeof(address));
18.     listen(server_fd, 3);//监听
19.     std::cout << "Server is listening on port 8080" << std::endl;
20.
21.     new_socket = accept(server_fd, (struct sockaddr *)&address, (sock
    len_t*)&addrlen);//验证有没有成功连接
```

Client 端：

```
1.  WSADATA wsaData;
2.  SOCKET clientSocket;
3.  struct sockaddr_in server;
4.
5.  // Initialize Winsock
6.  WSAStartup(MAKEWORD(2, 2), &wsaData);
7.
8.  // Create socket
9.  clientSocket = socket(AF_INET, SOCK_STREAM, 0);
10. if (clientSocket == INVALID_SOCKET) {
11.     std::cerr << "Socket creation failed." << std::endl;
12.     return 1;
13. }
14.
15. // Setup server structure
16. server.sin_family = AF_INET;
17. server.sin_addr.s_addr = inet_addr("192.168.43.147");
18. server.sin_port = htons(8080);
19.
20. // Connect to server
21. if (connect(clientSocket, (struct sockaddr*)&server, sizeof(server))
    == SOCKET_ERROR) {
22.     std::cerr << "Connection failed." << std::endl;
23.     closesocket(clientSocket);
24.     return 1;
25. }
26.
27. std::cout << "Connected to server." << std::endl;
```

Server 端（Linux）：

创建一个 TCP 套接字 server_fd，并检查是否成功创建。如果失败，输出错误信息并退出。

设置服务器地址结构 address，指定使用 IPv4 地址族 (AF_INET)，监听所有可用 IP 地址 (INADDR_ANY)，并将端口号设置为 8080。

使用 bind 函数将套接字与服务器地址绑定，以便监听特定端口。

使用 listen 函数开始监听客户端连接请求，同时指定最大连接数为 3。

通过 accept 函数接受客户端连接请求，建立新的套接字 new_socket，并返回客户端的地址信息。这一步用于验证连接是否成功。

Client 端（Windows）：

初始化 Winsock 库，以便在 Windows 上使用套接字通信。

创建一个 TCP 套接字 clientSocket，并检查是否成功创建。如果失败，输出错误信息并返回。

设置服务器地址结构 server，指定使用 IPv4 地址族 (AF_INET)，并指定服务器的 IP 地址和端口号为 8080。

使用 connect 函数尝试连接到服务器。如果连接失败，输出错误信息并关闭套接字。

如果连接成功，显示 "Connected to server."。

4.3.2 求和、求最大值结果传输

Server 端：

```
recv(new_socket, (char*)&bag, sizeof(bag), 0);
```

Client 端：

```
send(clientSocket, (char*)&bag, sizeof(bag), NULL);
```

简单传输，不过多赘述。

4.3.3 排序结果传输

Server 端：

```
void recvsort(int socket, float* remoteData) {
    int len;
    len = DATANUM;

    const int block_len = BUFFER_SIZE / sizeof(float);
    int recv_len = 0;
    int recv_size;
    while (recv_len < len) {

        recv_size = std::min(block_len, len - recv_len);
        ssize_t bytesRead = recv(socket, (void*)(remoteData + recv_le
n), recv_size * sizeof(float), 0);

        if(bytesRead <= 0){
            break;
        }

        recv_len += BUFFER_SIZE / sizeof(float);
```

```

    }


    // for (int i = 0; i < DATANUM / BUF_SIZE; i++) {
    //     retVal = recv(socket, buffer, BUF_SIZE, 0);

    //     startindex = i * BUF_SIZE / sizeof(float); // 调整为浮点数索引
    //     endindex = startindex + BUF_SIZE / sizeof(float);
    //     k = 0;

    //     for (int j = startindex; j < endindex; j++) {
    //         memcpy(temp, &buffer[k * sizeof(float)], sizeof(float));//从 buffer 中复制数据到 temp
    //         remoteData[j] = atof(temp);//将以 null 结尾的字符串转换成浮点数
    //         k++;
    //     }

    //     //发送确认信息，否则出现一组数没传完就连着传另一组数了，但是会增加耗时
    //     const char ack[] = "ok";
    //     send(socket, ack, sizeof(ack), 0);

    //     if (std::floor((i)/double(DATANUM/BUF_SIZE)*100)!=now){
    //         now = std::floor((i)/double(DATANUM/BUF_SIZE)*100);
    //         std::cout << "已收到"<< now << "%数据" << std::endl;
    //     }

    // }
}
```

Client 端：

```
void sendArray(int socket, float* data, const int len) {

    const int block_len = BUFFER_SIZE / sizeof(float);
    int send_len = 0;
    int send_size;
    while (send_len < len) {
        send_size = block_len < (len - send_len) ? block_len : len - send_len;
        size_t bytesSent = send(socket, (char*)(data + send_len), send_size * sizeof(float), 0);

```

```
        send_len += bytesSent / sizeof(float);
        //std::cout << "while 内
   " << "len=" << len << " recv_len=" << send_len << std::endl;
    }
}
```

这两个函数目的是从给定的套接字 socket 中接收一定数量的来自 Client 端的浮点数数据，并将这些数据存储到名为 remoteData 的浮点数数组中。代码使用循环逐块接收数据，直到总接收长度达到预定的数据长度 len 或者出现错误。

TCP 传输不能直接使用 send 和 recv 函数传递整个数组，因为 TCP 是面向流的传输协议，它并不了解发送的数据的语义结构。TCP 将数据视为连续的字节流，而不是分割成离散的消息或数据块。因此，直接传递整个数组会导致以下问题：

（1）粘包问题：由于 TCP 协议将数据视为字节流，多次连续的 send 操作可能会将多个消息或数据块合并成一个，这被称为粘包。接收方难以区分何时一个消息结束和下一个消息开始。

（2）拆包问题：相反，多次连续的 recv 操作可能会将一个消息拆分成多个部分，这被称为拆包。接收方需要额外的逻辑来重新组装拆分的数据，以还原原始的消息或数据块。

（3）缓冲区管理：直接传递整个数组需要大的接收缓冲区来存储可能不定长度的数据块，这可能会浪费内存或导致缓冲区溢出。

注释掉的代码是答辩时的代码，现在已经进行了修改。

相比之下，现在的函数在发送、接受数据时采用了更加紧凑和高效的方式，通过适当设置块大小，减少了数据复制和转换的次数，并且 while 循环实测下来比 for 循环在此功能实现上更高效。

五、测试结果

5.1 实验结果与数据

本次实验的条件如下表：

设备	笔记本 A	笔记本 B
CPU	13th Gen Intel(R) Core(TM) i9-13980HX	11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
系统	Linux Ubuntu20.04	Windows 11 专业版
通信角色	Server	Client

不加速实验和加速实验中都采取了双机个跑一半数据（64*1000000），再在 Server 端合并，结果如下表：

	序号	不加速时间（s）	加速时间（s）	加速比
求和	1	0.819186	0.00531949	153.997
	2	1.02412	0.00537753	190.444
	3	0.818667	0.00533562	153.434
	4	0.937899	0.00543577	172.542
	5	0.826505	0.00533072	155.046
	平均	0.8852754	0.005359826	165.0926
最大值	1	0.716555	0.0055858	128.281
	2	0.634696	0.00561056	113.125
	3	0.613926	0.0056035	109.561
	4	0.49541	0.00552644	89.6436
	5	0.60794	0.0056125	108.319
	平均	0.6137054	0.00558776	109.78592
排序	1	125.24	79.7873	1.56967
	2	130.212	113.021	1.1521
	3	154.742	112.547	1.37491
	4	134.752	106.143	1.26953
	5	119.279	84.3799	1.41359
	平均	132.845	99.17564	1.35596

5.2 结果分析

实验中，求和和求最大值的运算的加速比非常大。主要是由于 sse 加速的作用，同时对 8 个数进行运算，并且针对耗时很大的 sqrt 运算进行了优化，使得加速运算速度非常快。

而排序加速的效果并不理想，主要是因为传输过程的耗时过于大。虽然经过答辩之后，对网络传输的算法进行了优化，但是可能是因为手机热点连接传输的方式，带宽太小，耗时还是太多。

如果要进一步优化，一方面是采用线程池来管理线程，进行进一步的优化，另一方面是 sse 加速针对排序过程中更多地使用，进行同时多个数的排序。

当然如果使用网线进行传输，将大大缩减排序传输耗时，提高加速比。

5.3 复现说明

见提交文件中的 REAMME。

六、心得体会

本次作业因为种种原因，选择了一个人完成。虽然花了非常多的时间和精力，但是最后成果依然有诸多瑕疵，不令人满意。也让我得到了一个宝贵的教训，小组合作的意义不仅仅是分配工作，更重要的是可以有人在你陷入思维定势的时候及时指出，防止在错误的道路上越走越远，越走越歪。

比如一个非常简单，却令我刻骨铭心的错误。我一开始的不加速计算求和以及最大值的时候用时格外多，足足七八秒，导致最后的加速比大的不合常理。我花了好几天百思不得其解，甚至把简单的程序改的面目全非有没有对原程序进行备份。结果就是 Visual Studio 中我用 x86 运行。因为平时一直用 vscode，采用 cmake 的方式运行代码，结果在这个简单问题上越走越歪，得不偿失。

但是自己独立完成项目，也让我感到自己的代码习惯很糟糕，没有用.h 等文件存放功能函数等，之后要多向优秀的开源项目学习。