同济大学

TONGJI UNIVERSITY

《算法》大作业报告

报告名称	分布式并行计算实验报告
成员/分工	徐意 / 加速模块+报告 1, 2, 3, 5 部分
	宋睿轩 / 通信模块+报告 4, 6, 7 部分
任课教师	龚炜
日　　期	2025 年 12 月 20 日

分布式并行计算实验报告

1. 实验分析

本实验为算法课程的“分布式并行计算实验”，要求使用 C++编程语言，结合特定的加速技术和通信技术，高速地实现三个核心函数：数组求和、求最大值和排序，以处理千万上亿级别数量的浮点型数据。函数需要在两台计算机的协作下执行，旨在通过合理的设计尽可能挖掘双机的潜在算力，加之其他加速方法提升计算效率。

具体而言，规定可使用的加速技术包括 SSE 指令集、多线程、多进程和 OpenMP 等并行技术；通信技术包括 HTTP 和 Socket；操作系统平台包括 Windows、Linux（Ubuntu）和 Android。

2. 系统架构

本组采用 Ubuntu 操作系统作为实验平台，选取 Socket 的 UDP 协议作为多机通信方案，使用了 SSE 指令集和 OpenMP 和多机并行技术对核心函数进行加速处理。具体架构描述如下：

（1） 项目文件结构

本项目使用经典的 src、include、build 和 CMakeLists.txt 的 cmake 结构组织代码文件，利用头文件 common.h 链接各.cpp 文件、network_config.h 配置通信的 IP 和端口。其中.cpp 文件包括 basic.cpp、common.cpp、main.cpp、speed_up.cpp 和 UDP.cpp 共五个文件，说明如下：主函数位于 main.cpp 中作为 UDP 客户端与服务器端的入口，进入后执行不同的方案；basic.cpp 和 speed_up.cpp 分别实现了基础和加速版本的求和、求最大值、排序函数；UDP.cpp 负责通信模块的所有内容，common.cpp 包含了共用的数据初始化函数以及洗牌函数。

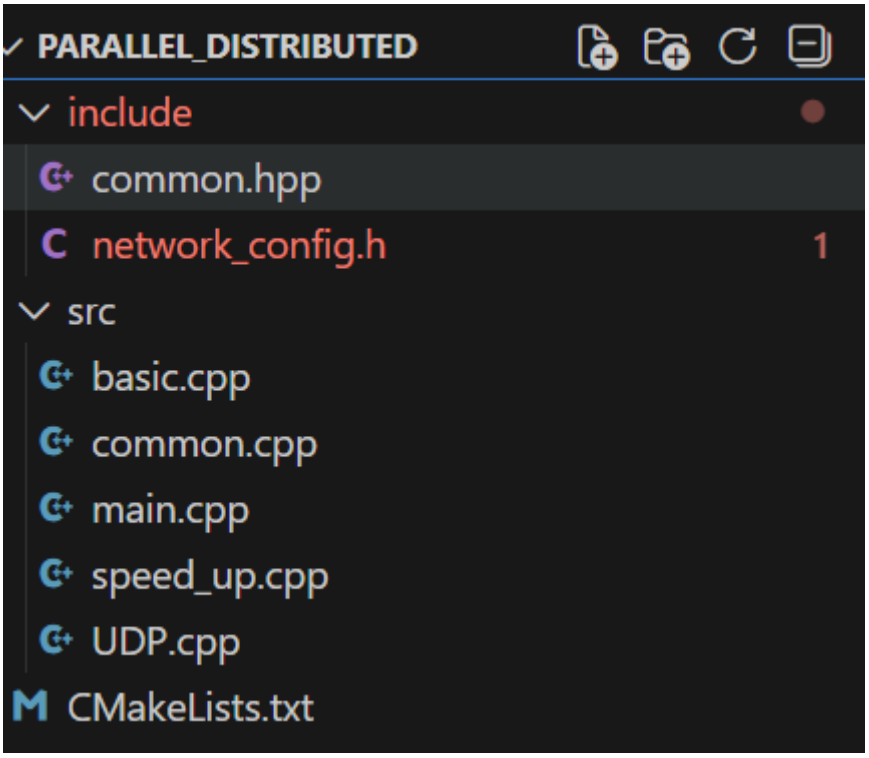

图 1 项目架构

（2） 加速方案架构

1. 各机器原始数组初始化分别在两个计算机完成并确保值域不发生重叠；
2. 两端分别使用 OpenMP 多线程同时处理多个浮点型数据（包括求和、求最大值、排序）；
3. 双机完成各自数据的处理后，通过 UDP 协议将数据发送到一端上合并，得到最终的处理结果和加速后总时长。

3. 任务划分策略

本组任务划分较为特殊。本组成员设备为一台 ubuntu 系统和一台 ubuntu 虚拟机，性能差异较大，因此在多机并行处理上对任务量的划分非平均。实测显示，ubuntu 系统处理数据的速度约为 ubuntu 虚拟机的 5 倍，根据多次测试经验值，分机加速处理部分数据量划分为：Server 端用 ubuntu 系统，处理 0.85 * DATANUM 的数据；Client 端用 ubuntu 虚拟机，处理剩下 0.15 * DATANUM 的数据，基本能保障两端同时完成处理。

若在同种设备运行，建议将比例调至 0.5。

4. 通信模块

4.1. 模块定位和功能概述

负责实现分布式并行计算中的双机通信，采用 UDP 协议完成两台计算机间的数据与指令交互。该模块为 sumSpeedUp、maxSpeedUp、sortSpeedUp 三个分布式加速函数提供底层通信支持，实现任务分发、远程计算、结果回传与合并。

4.2. 通信协议设计

自定义简单协议，消息格式包括操作类型（sum/max/sort）、数据长度、数据块索引等元信息。

4.3. UDP 通信实现：

主机端：创建 UDP 套接字，向从机发送任务请求，等待从机返回结果，超时重发。

从机端：监听端口，接收主机请求，解析后调用本地基础函数处理数据，结果通过 UDP 返回主机。

4.4. 代码调用与数据流关系图

本模块的核心通信与调度流程由 run_server()、run_client() 和 receive_thread(void* arg) 三个函数协作完成，具体如下：

1) run_server()

服务器端负责初始化 UDP 套接字并绑定本地端口，启动接收线程（receive_thread），等待客户端发送测试轮数（run_times），收到后进入多轮测试主循环。每轮测试分为两阶段：

基础版：仅 Server 本地处理全部数据，依次调用 sumBasic、maxBasic、sortBasic，计时并输出结果。处理完成后通过 UDP 通知 Client。

加速版：Server 和 Client 各自处理本地数据块。Server 初始化自己的数据，调用 sumSpeedUp、maxSpeedUp、sortSpeedUp 处理前半部分。等待 Client 通

过 UDP 返回 sum/max 结果，Server 本地与 Client 结果合并（排序结果合并可选实现）。

所有轮次结束后，统计并输出平均用时和加速比。

2) run_client()

客户端负责初始化 UDP 套接字，配置目标服务器地址，启动接收线程（receive_thread），由用户输入测试轮数，发送给 Server。每轮测试分为两阶段：

基础版：Client 仅等待 Server 处理完成（接收 BASIC_DONE 信号），收到后回复确认。

加速版：Client 初始化自己的数据（后半部分），调用 sumSpeedUp、maxSpeedUp、sortSpeedUp 处理本地数据。将 sum/max 结果通过 UDP 发送给 Server，最后发送 RESULTS_READY 信号。

所有轮次结束后关闭套接字。

3) void* receive_thread(void* arg)

独立线程，负责异步接收 UDP 消息，解析并处理：
识别控制信号（如 BASIC_DONE、RUN_TIMES、RESULT_SUM、RESULT_MAX、RESULTS_READY），更新全局状态变量，驱动主流程推进。支持多种消息类型，保证 Server 与 Client 间的同步与结果传递。

4) 计时说明

在 run_server 和 run_client 的每一轮测试中，均严格按照实验要求进行高精度计时。

基础版计时点：分别在 sumBasic、maxBasic、sortBasic 调用前后获取时间，统计单项和总用时。

加速版计时点：在 Server/Client 各自数据初始化后，调用 sumSpeedUp、maxSpeedUp、sortSpeedUp 前后获取时间，统计本地处理与最终合并的总用时。Server 端的加速版计时范围涵盖了任务分发、远程执行、结果收集与合并

全过程。

所有计时均采用高精度计时器（ clock_gettime/C++ chrono），并在多轮测试后取平均值，确保实验数据真实有效。

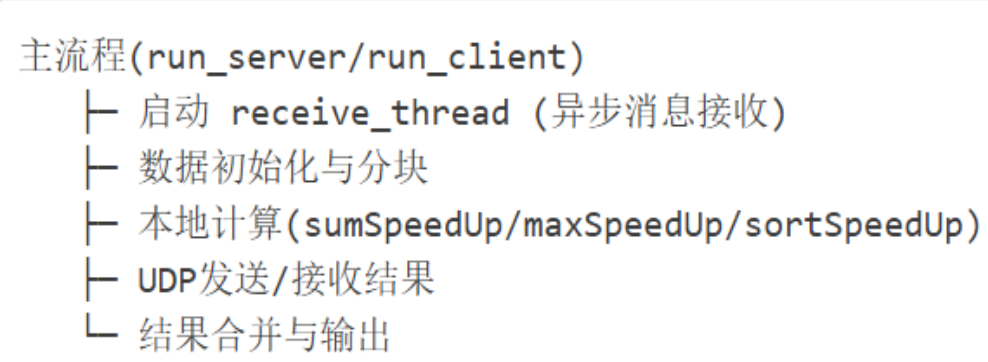

图 2 通信模块流程简图

5. 并行与向量化方法

单机的加速版本处理主要采用 OpenMP 多线程与 SSE 加速，具体如下。

1) float sumSpeedUp(const float data[], const int len)

在初始化 total_sum 之后使用#pragma omp parallel reduction(+:total_sum)进行线程分配，系统将根据设备自动生成一组合适数量的线程，对后续大括号内的内容并行处理，#pragma omp for nowait 将后续的计算循环分配到线程中；在每个循环内使用 SSE 加速计算 sqrt(x)，即每次使用__m128 vec_data 同时将四个浮点数转换成向量，然后使用__m128 v = _mm_sqrt_ps(vec_data);同时计算四个浮点数的开方；由于 log 没有 SSE 加速，只能再将向量 v 转换成标量求 log 和。

2) float maxSpeedUp(const float data[], const int len)

同上。

初始化 float global_max = -∞ 设为负无穷，保证任何有效值均可覆盖；#pragma omp parallel reduction(max:global_max)系统根据设备自动生成一组合适数量的线程，对后续大括号内的内容并行处理；#pragma omp for nowait 将主循环分配到线程中；线程完成自己的块后立即进入尾部循环，无 barrier 等待。

SSE 加速段（4 的倍数部分）：

_mm_loadu_ps(&data[i]) 一次加载 4 个 float 组成 128-bit 向量；_mm_sqrt_ps(vec_data) 同时计算 4 个开方；向量转标量后依次求 std::log，4 次比较更新当前线程私有 global_max；#pragma omp for nowait 再次无阻塞分配；每个线程独立处理残差元素并更新私有最大值；并行区结束时，OpenMP 自动对全部线程的 global_max 副本执行二叉树 max 归约，得到全局最大值。

3) void sortSpeedUp(const float data[], const int len, float* result)

初始化：

size_t* sortIndices = new size_t[len]; 创建索引数组；

#pragma omp parallel for 并行初始化，每个线程负责一段区间，避免单线程 memset 瓶颈。

线程分配：

外层 #pragma omp parallel 生成线程池；

#pragma omp single 仅由一条线程生成 root task，其余线程进入任务窃取循环。

任务并行递归（mergeSortParallel 函数）：

当区间长度 > max_threads * 2048 且深度 < 10 时，#pragma omp task 立即 fork 左右子区间任务，递归继续二分；任务粒度 ≤32 时回退到 插入排序，小数组用单线程快速完成；taskwait 只保证左右子区间完成，不阻塞其他任务，线程池始终满载。

并行归并（mergeParallel 函数）：

区间 > 8192 且深度 < 3 时，再次用 task 把合并阶段拆成两段并行 merge；二分查找分割点，保证左右子合并工作量均衡；最底层合并仍为串行，避免 task 开销大于计算。

结果填充：

#pragma omp parallel for schedule(static, block)按 4 线程分块 把排序后索引映射到结果数组，result[i] = std::log(std::sqrt(data[sortIndices[i]]))；

6. 实验结果

========== Round 1/5 ==========

[Basic 版本 - 只用 Server 端处理]

……

本轮 Basic 版本用时: 146213.25 ms

……

Speedup 版总共用时: 16254.65 ms（未单独统计各部分时间）

========== Round 2/5 ==========

[Basic 版本 - 只用 Server 端处理]

……

本轮 Basic 版本用时: 152594.88 ms

……

Speedup 版总共用时: 15094.89 ms （未单独统计各部分时间）

========== Round 3/5 ==========

……

本轮 Basic 版本用时: 144001.80 ms

……

Speedup 版总共用时: 16136.54 ms （未单独统计各部分时间）

========== Round 4/5 ==========

……

本轮 Basic 版本用时: 157620.58 ms

……

Speedup 版总共用时: 15531.76 ms （未单独统计各部分时间）

========== Round 5/5 ==========

……

本轮 Basic 版本用时: 145785.73 ms

……

Speedup 版总共用时: 15870.14 ms （未单独统计各部分时间）

==========

测试完成！统计信息:

==========

Basic 版本平均用时: 149243.25 ms

Speedup 版本平均用时: 15777.60 ms

加速比: 9.46x

==========

上述结果的 portion_server 值为 0.85。

7. 代码复现说明

7.1. 编译前的头文件配置

◆ IP 地址

文件：network_config.h

行号：第 9 行

说明：修改 SERVER_IP 为实际服务器 IP

#define SERVER_IP "127.0.0.1" // **可修改为实际服务器 IP**

◆ 端口号

文件：network_config.h

行号：第 15 行

说明：修改 SERVER_PORT 为所需端口号

#define SERVER_PORT 9999 // **可修改为实际端口号，建议 9999**

◆ 数据分配比例（portion_server）

文件：common.hpp

行号：第 16 行

说明：portion_server 控制服务器端处理数据的比例，客户端自动处理剩下的。

可根据两台电脑的性能分配调整：

如果两台电脑性能相近，建议设置为 0.5（各自处理一半数据）。

如果服务器性能更强，可适当提高此值（如 0.85），让服务器多处理一些数据。

#define portion_server 0.85 // **可修改比例以适配设备，性能相同时建议 0.5**

7.2. 编译和运行步骤

1. 在 linux 系统下打开终端
2. 进入该工程 build 目录
3. 生成构建文件（需已安装 CMake）：

   cmake ..
4. 编译项目：

   make
5. 运行程序：

   ./pardist
6. 根据提示输入数值以正确运行