中山大学计算机院本科生实验报告
（2023 学年春季学期）

课程名称：并行程序设计　　批改人：

实验	3-Pthreads 并行矩阵乘法与数组求和	专业（方向）	计算机科学与技术计科一班
学号	21307099	姓名	李英骏
Email	liyj323@mail2.sysu.edu.cn	完成日期	2024 年 4 月 28 日

目录

1 实验目的 **2**

2 实验过程和核心代码 **2**

2.1 矩阵乘法 3

2.1.1 main.cpp 3

2.1.2 Pthread_Matmul.cpp / .hpp 4

2.1.3 性能测试 7

2.2 数组求和 8

2.2.1 main.cpp 8

2.2.2 IntArray 11

2.2.3 Linear Aggregation 12

2.2.4 Tree Aggregation 12

3 实验结果 **13**

3.1 矩阵乘法 13

3.2 数组求和 15

4 实验感想 **17**

1 实验目的

1. 使用 Pthreads 实现并行矩阵乘法，并分析其性能。(尝试不同数据/任务划分方式)
2. 使用 Pthreads 实现并行数组求和，并分析其性能。(可分析不同聚合方式的影响)

2 实验过程和核心代码

矩阵乘法中尝试了两种数据划分方式:

1. 划分 1: 将 A 按行均分, 在每个线程中与完整的 B 相乘, 得出结果矩阵的部分行
2. 划分 2: 将 A 按行均分,B 按列均分, 在每个线程中相乘, 得出结果矩阵的一块

数组求和中尝试了两种聚合方式:

1. 所有线程算出结果后, 在主线程中统一求和
2. 树形规约

代码文件结构如图:

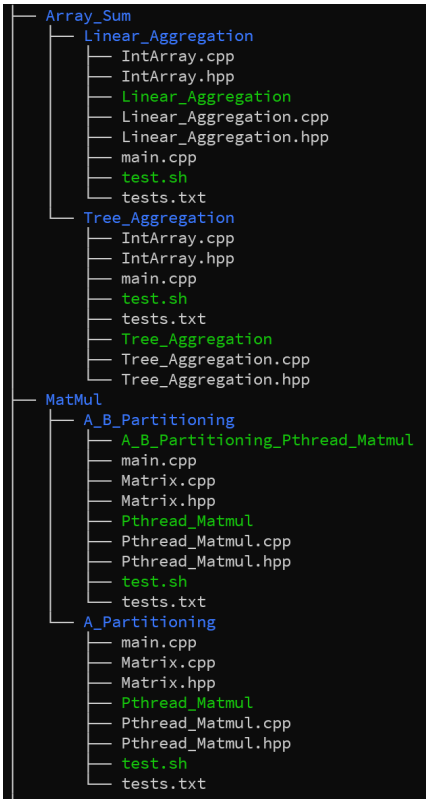


图 1: main.cpp

2.1 矩阵乘法

2.1.1 main.cpp

MatMul > A_Partitioning > main.cpp > main(int, char * [])

```cpp
#include "Pthread_Matmul.hpp"
#include <chrono>
#include <stdlib.h>

long int thread_count;
int m, n, k;
Matrix A, B, C;
int main(int argc, char *argv[])
{

    // 参数数量不正确则在主线程中打印报错并return
    if (argc != 5) {
        std::cerr << "Usage: " << argv[0] << " m n k" << std::endl;
        return 1;
    }

    // 读取矩阵参数
    thread_count = std::strtol(argv[1], NULL, 10);
    m = std::stoi(argv[2]);
    n = std::stoi(argv[3]);
    k = std::stoi(argv[4]);

    // 初始化矩阵
    A = Matrix(m, n, true);
    B = Matrix(n, k, true);
    C = Matrix(m, k, false);
    // 开始计时
    std::chrono::duration<double, std::milli> elapsed;
    auto start_time = std::chrono::high_resolution_clock::now();
    pthread_t *thread_handles = new pthread_t[thread_count];
    ThreadTasks thread_tasks(thread_count);
    // 计算矩阵乘法
    for (long thread = 0; thread < thread_count; ++thread) {
        pthread_create(&thread_handles[thread], NULL, pthread_matmul, (void *)&thread_tasks[thread]);
    }
    for (long thread = 0; thread < thread_count; ++thread) {
        pthread_join(thread_handles[thread], NULL);
    }
    auto end_time = std::chrono::high_resolution_clock::now();
    //  打印矩阵A B C
    std::cout << "Matrix A:\n";
    A.print_matrix();
    std::cout << "\nMatrix B:\n";
    B.print_matrix();
    std::cout << "\nMatrix C:\n";
    C.print_matrix();

    delete [] thread_handles;
    // 打印耗时
    elapsed = end_time - start_time;
    std::cout << "Time taken for matrix calculation: " << elapsed.count() / 1000 << " seconds\n";

    return 0;
}
```

图 2: main.cpp

详见注释.

2.1.2 Pthread_Matmul.cpp / .hpp

划分 1

MatMul > A_Partitioning > Pthread_Matmul.hpp > ...

```
1  #ifndef P_MATMUL
2  #define P_MATMUL
3
4  #include <pthread.h>
5  #include "Matrix.hpp"
6  extern Matrix A, B, C;
7
8  struct ThreadTask {
9      int row_start;
10     int row_end;
11 };
12
13 class ThreadTasks
14 {
15 private:
16     ThreadTask *thread_tasks;
17
18 public:
19     ThreadTasks() : thread_tasks(nullptr) {}
20     ThreadTasks(long int thread_count)
21     {
22         thread_tasks = new ThreadTask[thread_count];
23         int m = A.get_rows();
24         int rows_per_thread = m / thread_count;
25         for (long int thread = 0; thread < thread_count - 1; ++thread) {
26             thread_tasks[thread].row_start = thread * rows_per_thread;
27             thread_tasks[thread].row_end = (thread + 1) * rows_per_thread - 1;
28         }                                                  long thread_count
29         thread_tasks[thread_count - 1].row_start = (thread_count - 1) * rows_per_thread;
30         thread_tasks[thread_count - 1].row_end = m - 1;
31     }
32     ~ThreadTasks()
33     {
34         delete[] thread_tasks;
35         thread_tasks = nullptr;
36     }
37     ThreadTask& operator[](int index)
38     {
39         return this->thread_tasks[index];
40     }
41 };
42
43 void *pthread_matmul(void *arg);
44
45 #endif
```

图 3: Pthread_Matmul.hpp

注意 20-31 行, 记总线程数为 N, 三个矩阵分别为 $A_{m\times n}B_{n\times k}=C_{m\times k}$. 我们将矩阵 A 几乎平均地按行分成 N 份, 分发给 N 个 threads, 由于 A B C 是全局的, 我们只需要计算每个线程需要计算的行号, 传入即可.

矩阵乘法:

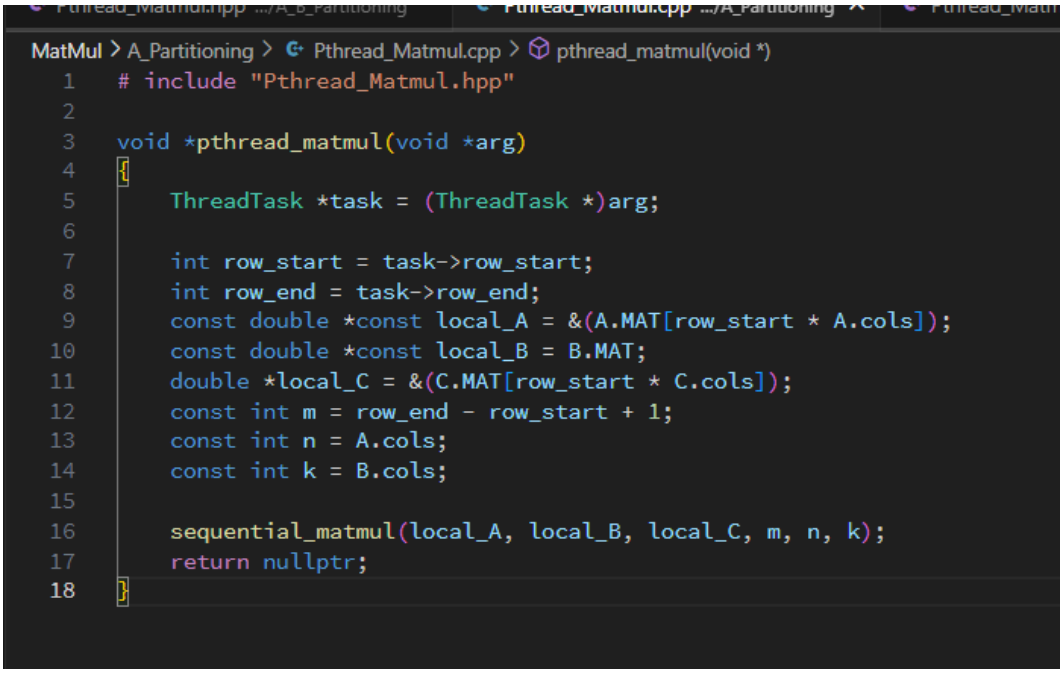


图 4: Pthread_Matmul.cpp

在每个线程中计算地址偏移, 再进行矩阵乘法即可.

划分 2

```cpp
ThreadTasks(long int thread_count)
{
    thread_tasks = new ThreadTask[thread_count];
    int m = A.get_rows();
    int k = B.get_cols();
    int A_partitions = 0;
    int B_partitions = 0;
    int sqrt_n = static_cast<int>(std::sqrt(thread_count + 0.1)); // 计算平方根并向下取整
    for (int factor = sqrt_n; factor >= 1; --factor) {
        if (thread_count % factor == 0) {
            A_partitions = thread_count / factor;
            B_partitions = factor;
            break;
        }
    }
    int A_rows_per_thread = m / A_partitions;
    int B_cols_per_thread = k / B_partitions;
    //std::cout << "MMM" << m << ' ' << A_partitions << ' ' << A_rows_per_thread << ' ' << (int)m / A_partitions << std::end
    //std::cout << "A B PER THREAD " << A_rows_per_thread << ' ' << B_cols_per_thread << std::endl;
    for (long int thread = 0; thread < thread_count; ++thread) {
        int i = thread / B_partitions;
        int j = thread % B_partitions;
        if (i == A_partitions - 1) {
            thread_tasks[thread].A_row_start = i * A_rows_per_thread;
            thread_tasks[thread].A_row_end = m - 1;
        } else {
            thread_tasks[thread].A_row_start = i * A_rows_per_thread;
            thread_tasks[thread].A_row_end = (i + 1) * A_rows_per_thread - 1;
        }

        if (j == B_partitions - 1) {
            thread_tasks[thread].B_col_start = j * B_cols_per_thread;
            thread_tasks[thread].B_col_end = k - 1;
        } else {
            thread_tasks[thread].B_col_start = j * B_cols_per_thread;
            thread_tasks[thread].B_col_end = (j + 1) * B_cols_per_thread - 1;
        }
    }
}
```

图 5: Pthread_Matmul.hpp

在第二种划分中, 我们将线程总数 N 拆分为 $N = a * b$, 将矩阵 A 按行分成 a 份,B 按列分成 b 份, 从而在每个线程中计算出 C 的一块.

```cpp
#include "Pthread_Matmul.hpp"
void *pthread_matmul(void *arg)
{
    ThreadTask *task = (ThreadTask *)arg;
    const int m = A.get_rows();
    const int n = A.get_cols();
    const int k = B.get_cols();
    std::cout << "A: " << task->A_row_start << ' ' << task->A_row_end << std::endl;
    std::cout << "B: " << task->B_col_start << ' ' << task->B_col_end << std::endl;
    const int A_row_start = task->A_row_start;
    const int A_row_end = task->A_row_end;
    const int B_col_start = task->B_col_start;
    const int B_col_end = task->B_col_end;
    partitioned_sequential_matmul(A.MAT, B.MAT, C.MAT, A_row_start, A_row_end, B_col_start, B_col_end, m, n, k);
    return nullptr;
}
```

图 6: Pthread_Matmul.cpp

其中 partitioned_sequential_matmul 的作用如图:

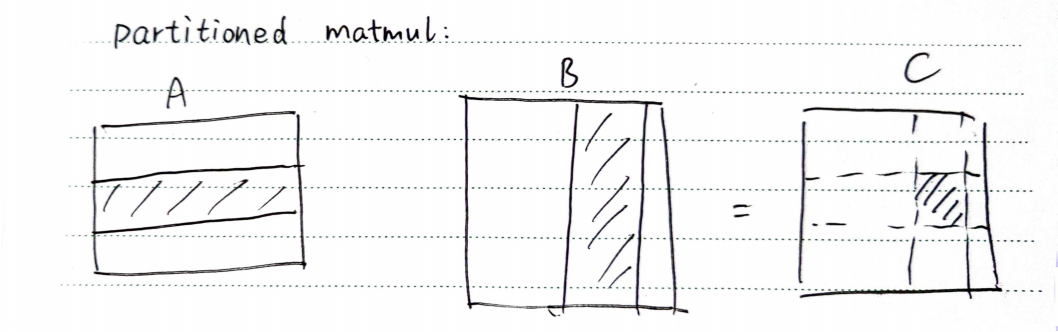


图 7: Pthread_Matmul.cpp

2.1.3 性能测试

我们使用如下脚本进行性能测试

MatMul > A_B_Partitioning > $ test.sh

```bash
#!/bin/bash

g++ -g -Wall -o A_B_Partitioning_Pthread_Matmul main.cpp Matrix.cpp Pthread_Matmul.cpp -lpthread

# 矩阵维度
declare -a inputs=("128 128 128" "256 256 256" "512 512 512" "1024 1024 1024" "2048 2048 2048")
# 进程数量
declare -a procs=(1 2 4 8 16)

for proc in "${procs[@]}"
do
    echo "worker线程数量: $((proc))"
    for input in "${inputs[@]}"
    do
        echo "    对于参数m,n,k: $input"
        total_time=0

        # 每组运行3次
        for i in {1..3}
        do
            echo "        运行 #$i"
            # 使用mpiexec运行
            output=$(./A_B_Partitioning_Pthread_Matmul $proc $input)

            time_taken=$(echo "$output" | grep -o -E 'Time taken for matrix calculation: [0-9.]+ seconds' | grep -o -E '[0-9.]+')
            echo "        time_taken: $time_taken"
            total_time=$(echo "$total_time + $time_taken" | bc)
        done

        average_time=$(echo "scale=6; $total_time / 3" | bc)
        echo "        平均运行时间: $average_time 秒"
    done
done

```

图 8: test.sh

2.2 数组求和

2.2.1 main.cpp

聚合 1

```cpp
#include "Linear_Aggregation.hpp"
#include <chrono>

long long thread_count;
long long N;
IntArray A;
int main(int argc, char *argv[])
{
    if (argc != 3) {
        std::cerr << "Usage: " << argv[0] << "Threads N" << std::endl;
        return 1;
    }
    thread_count = std::strtol(argv[1], NULL, 10);
    N = std::strtol(argv[2], NULL, 10);
    A = IntArray(N);
    // A.test();
    // 开始计时
    std::chrono::duration<double, std::milli> elapsed;
    auto start = std::chrono::high_resolution_clock::now();

    // 线程handles
    pthread_t *thread_handles = new pthread_t[thread_count];
    ThreadTasks thread_tasks(thread_count);
    for (long thread = 0; thread < thread_count; ++thread) {
        pthread_create(&thread_handles[thread], NULL, pthread_array_sum, (void *)&thread_ta
    }
    long long totalSum = 0;
    for (int i = 0; i < thread_count; ++i) {
        pthread_join(thread_handles[i], nullptr);
        totalSum += thread_tasks[i].result;
    }

    // 停止计时
    auto stop = std::chrono::high_resolution_clock::now();

    // 打印A 和 sum(A)
    std::cout << "Aarry A:\n";
    A.print();
    std::cout << "Sum:\n"
              << totalSum << '\n';

    // 检查求和结果是否正确
    /*long long realsum = A.partitioned_sum(0, A.get_size() - 1);
    std::cout << "Delta:\n"
              << realsum - totalSum << '\n';*/
    elapsed = stop - start;
    delete[] thread_handles;

    // 打印总耗时
    std::cout << "Time taken for matrix calculation: " << elapsed.count() / 1000 << " secor

    return 0;
}
```

图 9: main.cpp

聚合 2(树形聚合)

Array_Sum > Tree_Aggregation > main.cpp > ...

```cpp

long long thread_count;
long long N;
IntArray A;

// 全局数组和路障
std::vector<long long int> results;
pthread_barrier_t barrier;
int main(int argc, char *argv[])
{
    if (argc != 3) {
        std::cerr << "Usage: " << argv[0] << "Threads N" << std::endl;
        return 1;
    }
    thread_count = std::strtol(argv[1], NULL, 10);
    N = std::strtol(argv[2], NULL, 10);
    A = IntArray(N);
    // 全局数组,用于线程间通信
    results.resize(thread_count, 0);
    //A.test();
    // 开始计时
    std::chrono::duration<double, std::milli> elapsed;
    auto start = std::chrono::high_resolution_clock::now();

    // 线程handles
    pthread_t *thread_handles = new pthread_t[thread_count];
    ThreadTasks thread_tasks(thread_count);
    pthread_barrier_init(&barrier, NULL, thread_count);
    for (long long thread = 0; thread < thread_count; ++thread) {
        pthread_create(&thread_handles[thread], NULL, pthread_array_sum, (void *)&thread_tasks[thread]);
    }

    for (int i = 0; i < thread_count; ++i) {
        pthread_join(thread_handles[i], nullptr);
    }
    long long totalSum = results[0];
    // 停止计时
    auto stop = std::chrono::high_resolution_clock::now();

    // 打印A 和 sum(A)
    std::cout << "Aarry A:\n";
    //A.print();
    std::cout << "Sum:\n"
              << totalSum << '\n';

    // 检查求和结果是否正确
   /*  long long realsum = A.partitioned_sum(0, A.get_size() - 1);
    std::cout << "Delta:\n"
              << realsum - totalSum << '\n'; */
    elapsed = stop - start;
    delete[] thread_handles;

    // 打印总耗时
    std::cout << "Time taken for matrix calculation: " << elapsed.count() / 1000 << " seconds\n";

    return 0;
}

```

图 10: main.cpp

区别主要在于:

1. 树形规约为了实现进程间的同步和交流, 使用了一个全局数组和一个 barrier.

2. 线性规约需要对所有线程的 results 求和, 树形规约只要取数组的 0 号元素作为结果即可.

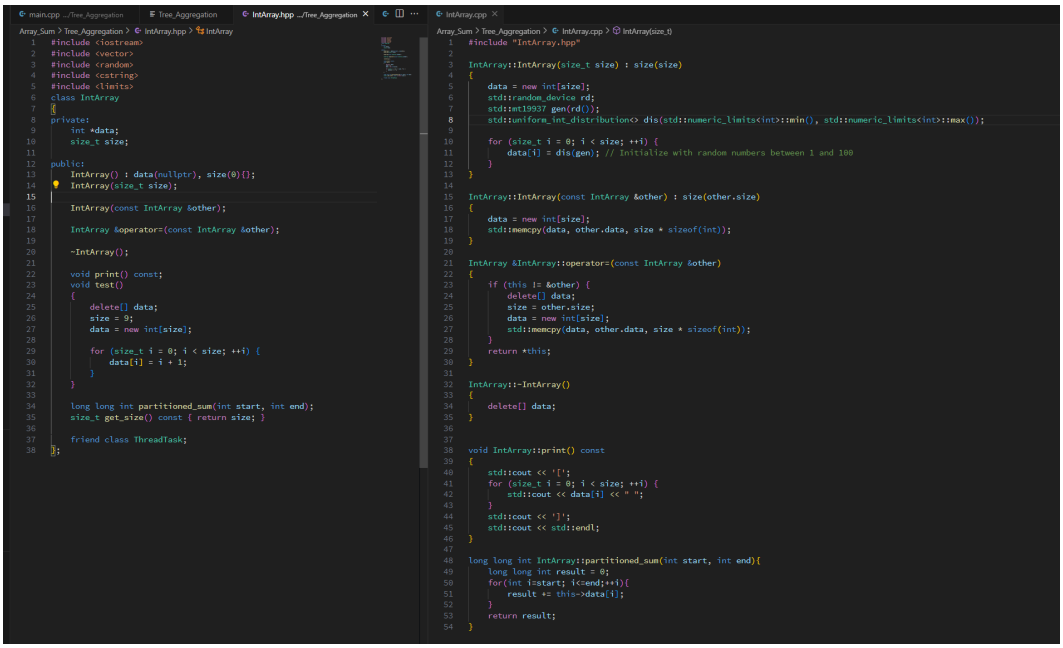
注释掉的部分用于检查算法正确性, 如图:

```
10 94 16 50 44 15 55 2 17 57 43 8 21 81 34 66 77 22 61 61 19 49 100 11 4
9 15 77 3 46 18 75 20 99 28 45 20 49 45 77 82 86 13 6 41 31 44 12 30 92
97 27 91 3 54 54 77 51 15 62 92 88 84 3 86 48 80 2 85 48 2 72 13 82 69 1
 65 32 92 14 75 57 54 29 16 23 73 35 18 20 41 44 48 60 28 28 1 9 75 46 9
6 67 32 52 15 89 57 64 71 76 42 98 53 87 27 5 29 93 2 89 91 84 20 2 19 8
5 79 87 66 29 96 46 58 13 47 81 65 34 81 76 66 11 36 69 80 81 1 88 68 7
58 68 23 23 41 36 51 6 80 41 53 95 74 43 66 43 82 2 27 27 61 75 79 15 91
 10 85 5 79 57 85 71 53 5 40 68 9 21 58 90 72 52 5 76 76 4 90 23 10 25 2
 82 75 76 28 70 58 8 17 25 55 91 27 10 8 63 66 14 86 85 21 85 51 99 5 53
 32 30 50 83 88 48 26 93 35 42 43 52 89 20 61 72 1 30 13 1 60 69 81 67 9
0 24 82 38 10 55 76 46 98 60 77 51 100 77 17 2 58 50 69 13 12 65 29 28 1
5 87 84 87 92 9 81 24 37 26 97 57 89 8 99 7 29 63 97 32 49 6 15 88 62 63
 65 27 51 3 71 16 30 77 6 77 51 96 27 87 55 73 96 24 94 4 29 23 85 79 28
 97 46 53 2 7 92 26 44 44 35 59 100 37 3 11 44 34 87 74 50 99 4 86 82 25
 69 49 63 68 78 6 65 45 74 94 89 90 76 56 2 69 67 23 87 25 91 30 49 60 6
9 58 43 8 70 71 81 36 61 31 24 36 36 5 27 12 43 16 61 3 7 75 79 44 21 50
 41 60 80 32 97 66 7 51 91 96 52 80 7 47 80 28 23 34 76 46 69 74 79 ]
Sum:
6471821
Delta:
0
Time taken for matrix calculation: 0.0014349 seconds
```

图 11: check

上图 Delta 为 0 表示并行求和的结果与单线程的循环求和结果一致, 说明算法正确.

2.2.2 IntArray



```hpp
#include <iostream>
#include <vector>
#include <random>
#include <cstring>
#include <limits>
class IntArray
{
private:
    int *data;
    size_t size;

public:
    IntArray() : data(nullptr), size(0){};
    IntArray(size_t size);

    IntArray(const IntArray &other);

    IntArray &operator=(const IntArray &other);

    ~IntArray();

    void print() const;
    void test()
    {
        delete[] data;
        size = 9;
        data = new int[size];

        for (size_t i = 0; i < size; ++i) {
            data[i] = i + 1;
        }
    }

    long long int partitioned_sum(int start, int end);
    size_t get_size() const { return size; }

    friend class ThreadTask;
};
```

```cpp
#include "IntArray.hpp"

IntArray::IntArray(size_t size) : size(size)
{
    data = new int[size];
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(std::numeric_limits<int>::min(), std::numeric_limits<int>::max());

    for (size_t i = 0; i < size; ++i) {
        data[i] = dis(gen); // Initialize with random numbers between 1 and 100
    }
}

IntArray::IntArray(const IntArray &other) : size(other.size)
{
    data = new int[size];
    std::memcpy(data, other.data, size * sizeof(int));
}

IntArray &IntArray::operator=(const IntArray &other)
{
    if (this != &other) {
        delete[] data;
        size = other.size;
        data = new int[size];
        std::memcpy(data, other.data, size * sizeof(int));
    }
    return *this;
}

IntArray::~IntArray()
{
    delete[] data;
}


void IntArray::print() const
{
    std::cout << '[';
    for (size_t i = 0; i < size; ++i) {
        std::cout << data[i] << " ";
    }
    std::cout << ']';
    std::cout << std::endl;
}

long long int IntArray::partitioned_sum(int start, int end){
    long long int result = 0;
    for(int i=start; i<=end;++i){
        result += this->data[i];
    }
    return result;
}
```

图 12: IntArray

第 3-13 行为初始化函数, 可以看到使用均匀分布函数, 将每个元素初始化为 int. 48-53 行为计算数组的部分和.

2.2.3 Linear Aggregation

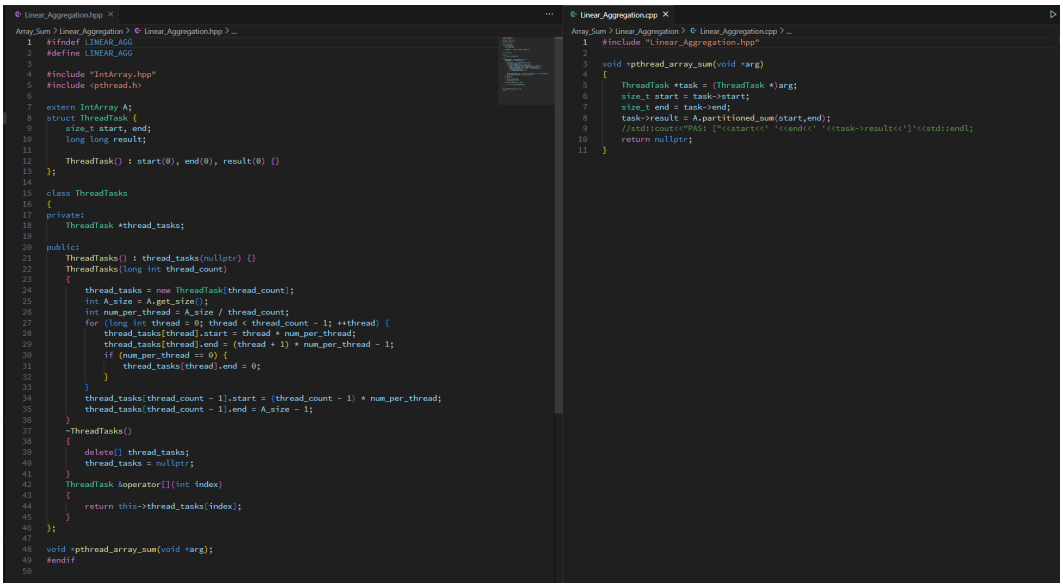


图 13: Linear Aggregation

简单计算部分和即可.

2.2.4 Tree Aggregation

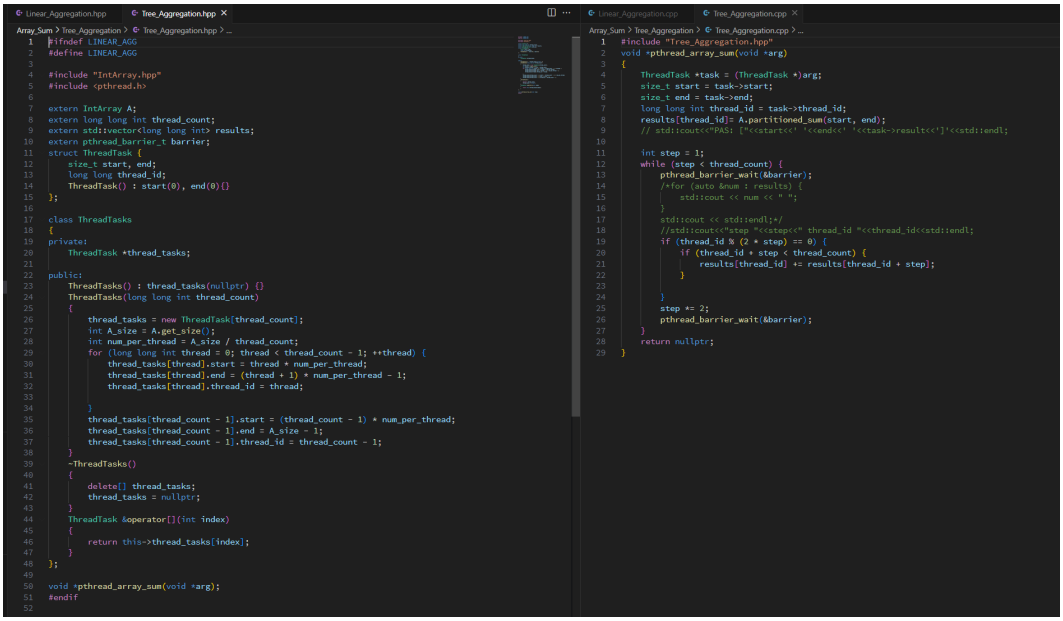


图 14: Tree Aggregation

右侧 11-27 行为树形规约, 为简便只考虑了线程数是 2 的幂次的情形. 实现了下图的二叉树:

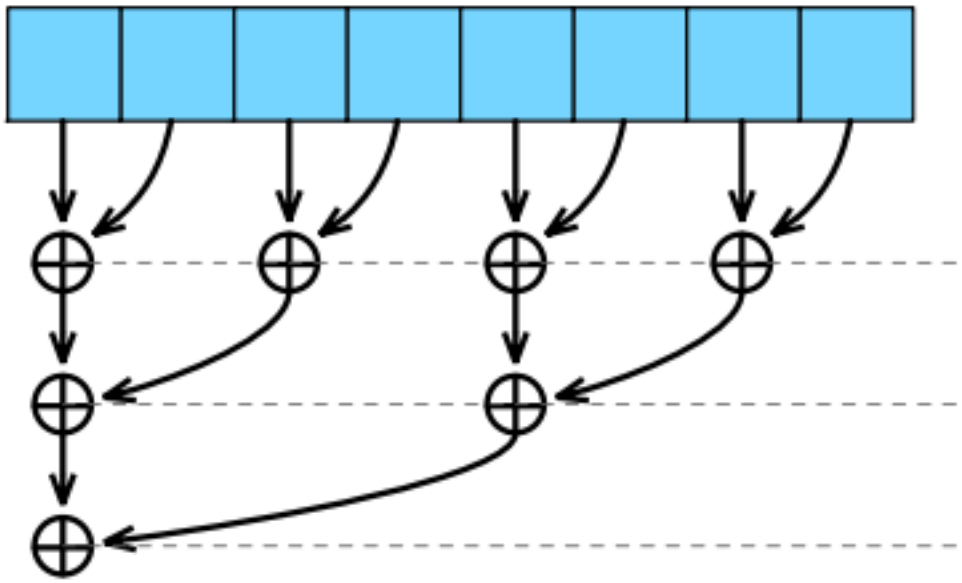


图 15: Tree Aggregation

3 实验结果

3.1 矩阵乘法

划分 1:

表 1: 划分 1　线程数与矩阵规模的性能数据 (秒)

矩阵规模 / worker 进程数	128	256	512	1024	2048
1	.003431	.026890	.219175	1.711090	13.725566
2	.002038	.014694	.109977	.853172	6.847813
4	.001028	.007010	.056423	.418382	3.421770
8	.000829	.004758	.041580	.243998	2.082076
16	.000796	.003455	.028399	.210076	1.474280

划分 2:

表 2: 划分 2　线程数与矩阵规模的性能数据 (秒)

矩阵规模 / worker 进程数	128	256	512	1024	2048
1	.003523	.026857	.210081	1.726030	13.360266
2	.001849	.013418	.106289	.843146	6.833443
4	.001564	.010202	.058136	.448828	3.569666
8	.001168	.007704	.031460	.243478	1.864370
16	.001078	.006368	.035039	.192912	1.255803

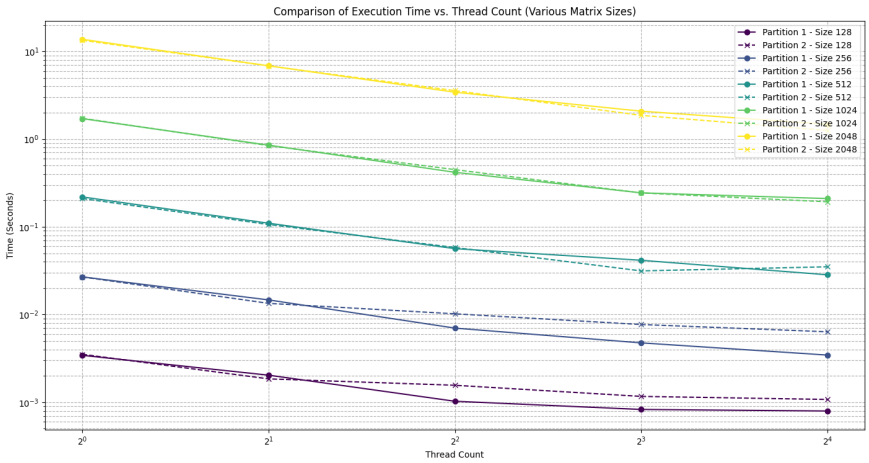


图 16: result

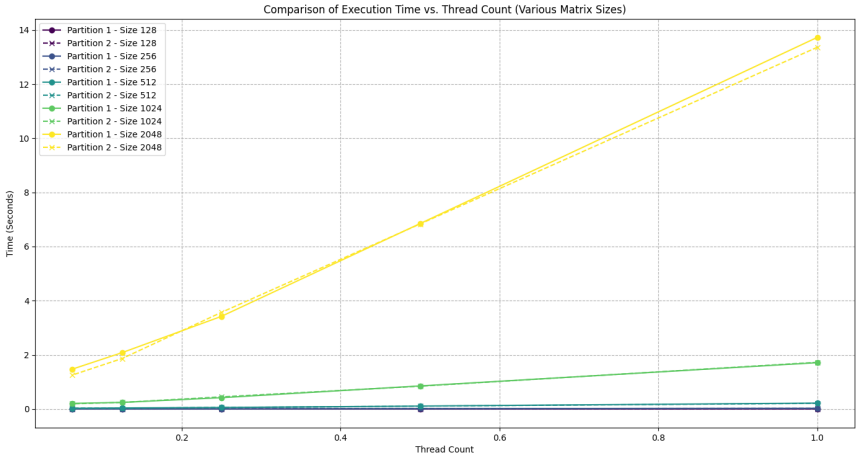


图 17: result

观察出以下结论:

1. 时间开销随着线程数量的增加几乎反比例地下降 (图中近似直线).

2. 两种划分算法性能相仿. 第二种算法在矩阵较大时略优于第一种算法.

3.2 数组求和

划分 1:

表 3: 聚合 1　线程数与数组规模的性能数据 (秒)

数组规模 (M) / worker 进程数	1	2	4	8	16	32	64	128
1	.001481	.002983	.005647	.011060	.021771	.044333	.088551	.178929
2	.000806	.001530	.002885	.005695	.011572	.023171	.047254	.092074
4	.000623	.000949	.001713	.004073	.007763	.014384	.023820	.045619
8	.000762	.001147	.001613	.003309	.006813	.013369	.013515	.041619
16	.001367	.001779	.002284	.004367	.005254	.011448	.018628	.033708

划分 2:

表 4: 聚合 2　线程数与数组规模的性能数据 (秒)

worker 进程数 \ 数组规模 (M)	1	2	4	8	16	32	64	128
1	.001505	.002958	.005740	.011594	.022237	.043872	.085593	.173632
2	.000893	.001481	.002779	.005510	.011183	.022117	.044044	.092859
4	.000681	.001270	.002966	.002889	.005716	.011321	.023251	.052073
8	.000629	.001289	.001648	.002874	.004138	.009129	.019201	.035495
16	.000517	.000816	.001202	.002754	.004697	.008007	.015590	.026738

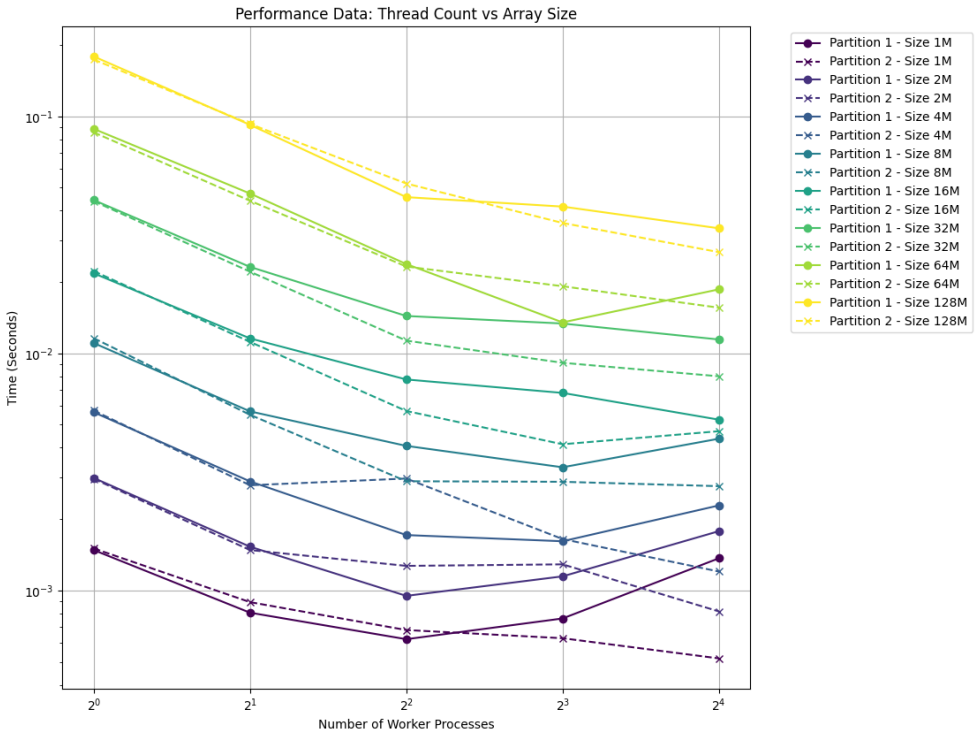


图 18: result

观察出以下结论:

1. 树形规约在线程数较多时, 显著优于线性规约, 而在线程数较少时, 由于 barrier 的影响, 可能劣于线性规约.

2. 数组维度较小时, 线程数增加反而会降低求和速度. 这是由于线程池创建和销毁线程的开销, 以及上下文切换的开销较大.

4 实验感想

通过本次实验，我深入了解了并行程序设计中矩阵乘法和数组求和的实现方式，特别是如何利用 pthread 实现高效的并行计算。实验中，我尝试了两种不同的数据划分方式，即按行均分和行列均分，这不仅加深了我对并行计算中数据划分策略的理解，也让我认识到在实际应用中，针对不同的计算任务选择合适的划分策略是多么重要。

同时, 我还了解了树形规约的实现方式.

此外，性能测试也是本次实验的重要部分。通过对比不同划分方式下的执行时间，我更加直观地理解了并行计算的性能优化