中山大學
SUN YAT-SEN UNIVERSITY

# 并行程序设计与算法实验

## Lab3-Pthreads 并行矩阵乘法与数组求和


姓 名 马岱

学 号 22336180

学 院 计算机学院

专 业 计算机科学与技术


2025 年 4 月 9 日

# 1 实验目的

- Pthreads 程序编写、运行与调试
- 多线程并行矩阵乘法
- 多线程并行数组求和

# 2 实验内容

- 掌握 Pthreads 编程的基本流程
- 理解线程间通信与资源共享机制
- 通过性能分析明确线程数、数据规模与加速比的关系

## 2.1 并行矩阵乘法

- 使用 Pthreads 实现并行矩阵乘法
- 随机生成 $m \times n$ 的矩阵 $A$ 及 $n \times k$ 的矩阵 $B$
- 通过多线程计算矩阵乘积 $C = A \times B$
- 调整线程数量 (1-16) 和矩阵规模 (128-2048)，记录计算时间
- 分析并行性能 (时间、效率、可扩展性)
- 选做：分析不同数据划分方式的影响
  - 请描述你的数据/任务划分方式。
  - 回答：采用二维分块策略将输出矩阵 C 分成若干子块，每个线程负责计算一个子块。设定线程数 $P$ 后，计算分块数目为行方向 $R = \lfloor \sqrt{P} \rfloor$ 和列方向 $C = \lceil P/R \rceil$；这样总块数 $R \times C$ 不小于线程数，将前 $P$ 个块分配给线程计算。对于每个块，依据行和列的均分加上余数进行调整，确定对应的起始和结束行、列。

### 2.1.1 代码思路

首先创建一个共享信息结构体和私有信息结构体，其中共享信息结构体中存放的是每个线程都共有的资源（矩阵及其大小），私有信息结构体中存放每个线程的需要执行的矩阵的行和列。

```c
// 共享信息结构体
typedef struct
{
    int m, n, k; // A 为 m×n, B 为 n×k, C 为 m×k
    double *A, *B, *C;
} share_info;

// 每个线程的私有信息
typedef struct
{
    share_info *info;
    int start_row;
    int end_row;
} thread_info;
```

由于矩阵运算访问的元素不存在读写冲突或写写冲突，所以不需要做特殊的互斥处理。这里直接定义线程调用的矩阵乘法函数。注意这里我先在外面定义了一个 double 类型的 sum，然后再进入循环计算，这里能够明显降低程序的开销。这实际上是一个让出缓存行的操作，可以看到，A[i * n + p] 在最里层的循环是不变的，但是这里却频繁地进行数组访问，如果是之前的进程还好，而线程就会导致 cache 中产生没有意义的重复占用，这里采用一个临时变量 (由于编译的优化会放入寄存器中)，改写内层代码。

```c
void *thread_func(void *arg)
{
    thread_info *tinfo = (thread_info *)arg;
    share_info *info = tinfo->info;
    int m = info->m;
    int n = info->n;
    int k = info->k;
    int start = tinfo->start_row;
    int end = tinfo->end_row;

    for (int i = start; i < end; i++)
    {
        for (int j = 0; j < k; j++)
        {
            double sum = 0.0;
            for (int p = 0; p < n; p++)
            {
                sum += info->A[i * n + p] * info->B[p * k + j];
```

```
            }
            info->C[i * k + j] = sum;
        }
    }
    return NULL;
}
```

在主函数内以下代码是主要的线程执行代码，其实没有什么过多需要强调的，但是这里有一点需要重点注意，那就是时间的计算。这里注意不能使用 ‘clock()‘来记录时间，因为 ‘clock()‘ 计算的是 ‘the CPU time used so far‘ ，即占用的 CPU 时间。而多线程和单线程不同的是，多线程会占用更多的 CPU 时间（多个线程同时运行），因此，多线程下使用 ‘clock()‘ 会造成结果过大。因此我们使用 ‘gettimeofday()‘来记录时间。

```
    // 记录开始时间
    struct timeval start_time, end_time;
    gettimeofday(&start_time, NULL);

    for (int i = 0; i < num_threads; i++)
    {
        tinfo[i].info = &info;
        tinfo[i].start_row = current_row;
        // 如果还有剩余行则均匀分配
        tinfo[i].end_row = current_row + rows_per_thread + (i < extra ? 1 : 0);
        current_row = tinfo[i].end_row;
        pthread_create(&threads[i], NULL, thread_func, (void *)&tinfo[i]);
    }
    // 等待所有线程完成工作
    for (int i = 0; i < num_threads; i++)
    {
        pthread_join(threads[i], NULL);
    }

    // 记录结束时间
    gettimeofday(&end_time, NULL);
```

最后为了运行方便，我还特意写了一个脚本，该脚本可以自动化测试不同矩阵规模和线程数下的并行矩阵乘法。

```
#!/bin/bash
# 可执行文件名称（编译后的程序名称）
executable="./pthread1"
```

```

gcc -o pthread1 pthread1.c -lpthread -lm
if [ $? -ne 0 ]; then
    echo " 编译失败，请检查代码。"
    exit 1
fi

# 定义正方形矩阵大小数组
matrix_sizes=(128 256 512 1024 2048)
# 定义线程数数组
thread_counts=(1 2 4 8 16)

# 定义日志文件
log_file="run_tests.log"
echo " 矩阵规模和线程数测试记录" > "$log_file"
echo " 测试日期: $(date)" >> "$log_file"
echo "----------------------------------" >> "$log_file"

# 对每种矩阵规模和线程数进行测试
for size in "${matrix_sizes[@]}"; do
    echo " 测试矩阵规模: ${size}x${size}" | tee -a "$log_file"
    for threads in "${thread_counts[@]}"; do
        echo "  线程数: ${threads}" | tee -a "$log_file"
        # 运行程序，并将输出存入变量 result
        result=$($executable $size $size $size $threads)
        echo "$result" | tee -a "$log_file"
        echo "----------------------------------" >> "$log_file"
    done
    echo "==================================" | tee -a "$log_file"
done

echo " 所有测试完成，详细日志请查看 $log_file"
```

### 2.1.2 选做：块划分

这里在原有的共享信息上仅修改私有信息即可。每个线程拥有自己的起始行、结束行、起始列和结束列。

```
// 每个线程的私有信息（用于 2D 块划分）
typedef struct
{
```

```
    share_info *info;
    int row_start, row_end; // 输出矩阵 C 的行区间 [row_start, row_end)
    int col_start, col_end; // 输出矩阵 C 的列区间 [col_start, col_end)
} block_thread_info;
```

这里采用块划分，和之前的行划分最不一样的就是最外层两重循环是为了遍历这个线程要负责的 C 的子矩阵区域，内部每个 (i, j) 元素需要通过一次内积计算得到。先将传入的 arg 强转为 block_thread_info *，再解析出需要计算的矩阵 C 的行、列区间 [rs, re) 和 [cs, ce)。

```
// 每个线程的工作函数，计算子块 C(row_start:row_end, col_start:col_end)
void *block_thread_func(void *arg) {
    block_thread_info *btinfo = (block_thread_info *)arg;
    share_info *info = btinfo->info;
    int m = info->m, n = info->n, k = info->k;
    int rs = btinfo->row_start, re = btinfo->row_end;
    int cs = btinfo->col_start, ce = btinfo->col_end;
    for (int i = rs; i < re; i++) {
        for (int j = cs; j < ce; j++) {
            double sum = 0.0;
            for (int p = 0; p < n; p++) {
                sum += info->A[i * n + p] * info->B[p * k + j];
            }
            info->C[i * k + j] = sum;
        }
    }
    return NULL;
}
```

## 2.2 并行数组求和

- 使用 Pthreads 实现并行数组求和
- 随机生成长度为 $n$ 的整型数组 $A$，$n$ 取值范围 [1M, 128M]
- 通过多线程计算数组元素和 $s = \sum_{i=1}^{n} A_i$
- 调整线程数量 (1-16) 和数组规模 (1M-128M)，记录计算时间
- 分析并行性能 (时间、效率、可扩展性)
- 选做：分析不同聚合方式的影响

- 请描述你的聚合方式。
- 回答：采用全局累加器配合互斥锁，由各线程直接更新全局变量，从而实现线程之间的即时聚合。

  原来的代码中，所有线程先各自计算局部和，最后在主线程中将各个局部和累加得到最终结果；而我们可以直接采用全局累加器配合互斥锁，由各线程直接更新全局变量，从而实现线程之间的即时聚合。

### 2.2.1 代码思路

这里和上面的矩阵乘法一样，首先创建一个共享信息结构体和私有信息结构体，其中共享信息结构体中存放的是每个线程都共有的资源（数组大小和数组本身），私有信息结构体中存放每个线程的需要执行的数组的下标。

```
// 共享结构体数组
typedef struct
{
    int n;
    int *num;
} share_info;

// 每个线程的私有信息
typedef struct
{
    share_info *info;
    int start_index;
    int end_index;
    long long local_sum;
} thread_info;
```

由于仅仅是数组求和，因此无论先求还是后求，元素都不存在读写冲突或写写冲突，所以不需要做特殊的互斥处理。这里直接定义线程调用的数组加法函数。计算每一部分的局部和，然后再将结果统一到结构体中的总 sum 中。

```
void *thread_sum(void *arg)
{
    thread_info *tinfo = (thread_info *)arg;
    share_info *info = tinfo->info;
    int n = info->n;
    int *num = info->num;
    long long sum = 0;

```

```
    // 计算局部和
    for (int i = tinfo->start_index; i < tinfo->end_index; i++)
    {
        sum += info->num[i];
    }
    tinfo->local_sum = sum; // 将局部和存储到结构体中
    return NULL;
}
```

在主函数内以下代码是主要的线程执行代码，其实没有什么过多需要强调的，时间的计算也同样需要注意，和上面的矩阵乘法一样需要使用 gettimeofday 来获取。

```
    struct timeval start_time, end_time;
    gettimeofday(&start_time, NULL);
    for (int i = 0; i < num_threads; i++)
    {
        tinfo[i].info = &info;
        tinfo[i].start_index = start_index;
        if (i < remainder)
        {
            tinfo[i].end_index = start_index + chunk_size + 1; // 多余的部分分
            ↪ 配给前几个线程
        }
        else
        {
            tinfo[i].end_index = start_index + chunk_size;
        }
        tinfo[i].local_sum = 0;
        start_index = tinfo[i].end_index;
        pthread_create(&threads[i], NULL, thread_sum, (void *)&tinfo[i]);
    }
    // 等待所有线程完成工作
    long long total_sum = 0;
    for (int i = 0; i < num_threads; i++)
    {
        pthread_join(threads[i], NULL);
        total_sum += tinfo[i].local_sum;
    }
    // 记录结束时间
    gettimeofday(&end_time, NULL); // 记录结束时间
```

### 2.2.2 选做：各线程直接更新全局变量

这里定义了全局变量 global_sum 作为所有线程共同的累加结果，并用 pthread_mutex_t sum_mutex 来保护该变量。

```
// 全局累加器以及互斥锁
long long global_sum = 0;
pthread_mutex_t sum_mutex = PTHREAD_MUTEX_INITIALIZER;
```

这里直接定义线程函数执行过程为计算区间内局部和后直接更新全局和，注意这里使用了 mutex 来加锁。

```
// 线程函数：计算区间内局部和后更新全局和
void *thread_sum(void *arg) {
    thread_info *tinfo = (thread_info *)arg;
    share_info *info = tinfo->info;
    long long local_sum = 0;
    for (int i = tinfo->start_index; i < tinfo->end_index; i++) {
        local_sum += info->num[i];
    }
    // 使用互斥锁更新全局累加和
    pthread_mutex_lock(&sum_mutex);
    global_sum += local_sum;
    pthread_mutex_unlock(&sum_mutex);
    return NULL;
}
```

# 3 实验结果

## 3.1 并行矩阵乘法

表 1: 并行矩阵乘法在不同线程数下的运行时间

| 矩阵规模 | 1 线程 | 2 线程 | 4 线程 | 8 线程 | 16 线程 |
|---|---|---|---|---|---|
| 128×128 | 0.01232 s | 0.00545 s | 0.00449 s | 0.00533 s | 0.01422 s |
| 256×256 | 0.06298 s | 0.04175 s | 0.03200 s | 0.02459 s | 0.02913 s |
| 512×512 | 0.60493 s | 0.29455 s | 0.15440 s | 0.16169 s | 0.14499 s |
| 1024×1024 | 9.63817 s | 2.73088 s | 1.28812 s | 1.20201 s | 1.34612 s |
| 2048×2048 | 101.69945 s | 41.56598 s | 20.11231 s | 19.29964 s | 20.05837 s |

### 3.2 并行数组求和

表 2: 数组求和不同线程数下的运行时间

| 数组规模 | 1 线程 | 2 线程 | 4 线程 | 8 线程 | 16 线程 |
|---|---|---|---|---|---|
| 1M | 0.00409s | 0.00269s | 0.00381s | 0.00516s | 0.00555s |
| 4M | 0.01443s | 0.00745s | 0.00685s | 0.00711s | 0.00853s |
| 16M | 0.05151s | 0.03615s | 0.02028s | 0.01554s | 0.01745s |
| 64M | 0.20709s | 0.11965s | 0.08471s | 0.07442s | 0.10135s |
| 128M | 0.44321s | 0.22185s | 0.10702s | 0.11298s | 0.12607s |

## 4 实验分析

### 4.1 并行矩阵乘法

表 3: 并行矩阵乘法加速比

| 矩阵规模 | 1 线程 | 2 线程 | 4 线程 | 8 线程 | 16 线程 |
|---|---|---|---|---|---|
| 128×128 | 1.000 | 2.261 | 2.744 | 2.312 | 0.865 |
| 256×256 | 1.000 | 1.509 | 1.968 | 2.562 | 2.163 |
| 512×512 | 1.000 | 2.055 | 3.915 | 3.743 | 4.171 |
| 1024×1024 | 1.000 | 3.529 | 7.480 | 8.022 | 7.163 |
| 2048×2048 | 1.000 | 2.447 | 5.058 | 5.268 | 5.071 |

表 4: 并行矩阵乘法效率

| 矩阵规模 | 1 线程 | 2 线程 | 4 线程 | 8 线程 | 16 线程 |
|---|---|---|---|---|---|
| 128×128 | 1.000 | 1.130 | 0.686 | 0.289 | 0.0541 |
| 256×256 | 1.000 | 0.7545 | 0.492 | 0.320 | 0.1352 |
| 512×512 | 1.000 | 1.0275 | 0.979 | 0.4679 | 0.2607 |
| 1024×1024 | 1.000 | 1.7645 | 1.870 | 1.0028 | 0.4477 |
| 2048×2048 | 1.000 | 1.2237 | 1.2645 | 0.6585 | 0.3169 |

- **线程数量对性能的影响分析**：我们发现随着线程数量增加，性能提升**并非简单呈线性趋势**。例如，对于 128×128 矩阵，单线程运行时间为 0.01232 秒，而 2 线程可降低到 0.00545 秒，取得加速比约 2.26；但当线程增加到 4、8 甚至 16 时，运行时间反而出现波动或上升，4 线程时为 0.00449 秒（加速比约 2.74），8 线程时

为 0.00533 秒（加速比约 2.31）。这一现象表明，随着线程数量增加，程序的运行时间逐渐减少，说明多线程可以有效地提高程序的性能。然而，多线程带来的额外开销可能会超过并行计算的收益，导致当线程数超过一定数值时，可以看到加速比并不是一直随着线程规模的增大而增大，由于**核数的限制**（我的电脑只有 4 核），加速比反而会有所**下降**。

- **矩阵规模对并行效率的影响**：实验数据表明，随着矩阵规模的增大，单线程计算时间显著增加，从而使得多线程计算能获得更充分的并行计算收益。以 1024×1024 矩阵为例，单线程运行时间为 9.638 秒，而 4 线程时下降到 1.288 秒，显示出**较高的加速比**；同样，2048×2048 矩阵的单线程运行时间高达 101.699 秒，通过多线程降低到 20 秒左右。然而，即使在大规模问题下，加速比依然没有达到理论上线程数增长带来的线性提升，这说明数据传输、内存带宽以及线程同步等因素依然会**限制并行效率**的提升。总体来看，矩阵规模越大，每个线程处理的数据量越多，计算开销相对线程管理的成本就越低，因而并行效率也有所提高，但硬件（核数）限制仍制约着效率的进一步改善。

- **可扩展性分析**：从整体实验结果来看，在**固定问题规模下**（即强可扩展性测试），增加线程数量在大规模矩阵下可以获得**较明显的加速**，但当线程数超过 8 个后，加速比开始趋于平缓甚至下降，这反映出系统受到内存带宽、缓存一致性及线程同步开销等因素的限制，表现为**部分强可扩展性**；对于小规模矩阵，增加线程数甚至会造成负扩展现象，从而属于无可扩展状态。在弱可扩展性方面，如果**同时按比例增加问题规模和线程数**，那么每个线程的负载保持不变，理论上可以保持运行时间恒定，但实际情况依然受到通信、汇总以及内存访问延迟的影响，**弱可扩展性**也**未能**达到理想状态。综合来看，在大规模数据下均呈现出有限的扩展性，整体性能**介于部分强可扩展和弱可扩展**之间。

- (选做) 不同数据划分方式的比较：

表 5: 块划分的运行时间

| 矩阵规模 | 1 线程 | 2 线程 | 4 线程 | 8 线程 | 16 线程 |
|---|---|---|---|---|---|
| 128×128 | 0.00970 s | 0.00890 s | 0.00551 s | 0.01283 s | 0.02620 s |
| 256×256 | 0.06955 s | 0.04071 s | 0.03196 s | 0.02010 s | 0.02468 s |
| 512×512 | 0.62114 s | 0.29889 s | 0.32209 s | 0.17662 s | 0.18708 s |
| 1024×1024 | 10.21989 s | 3.51717 s | 1.87348 s | 2.22072 s | 1.62429 s |
| 2048×2048 | 110.88003 s | 52.31170 s | 20.04063 s | 20.15265 s | 21.55310 s |

对于 128×128 矩阵，行划分和块划分的**性能差异较小**，单线程耗时分别为 0.01232 s 和 0.00970 s，且多线程下两种方法均受到线程调度开销影响，结果相近。对于

256×256 与 512×512 矩阵，两种方法在 1 线程和 2 线程下表现基本一致，而在 4 线程以上时，行划分略微领先，例如 512×512 矩阵在 8 线程下行划分耗时为 0.16169 s，而块划分为 0.17662 s。在大规模测试（1024×1024 及 2048×2048）中，行划分始终保持相对稳定的加速效果，如 1024×1024 矩阵 16 线程时耗时 1.34612 s，对比块划分的 1.62429 s；2048×2048 矩阵单线程时行划分耗时 101.69945 s，而块划分则略高（110.88003 s），16 线程时行划分耗时 20.05837 s，而块划分耗时 21.55310 s。总体来看，**行划分方法**由于**实现简单**，带来的额外管理开销更低，性能表现略优；**块划分**理论上有助于**改善缓存局部性**，但实际效果并不明显。

## 4.2 并行数组求和

表 6: 并行数组求和加速比

| 数组规模 | 1 线程 | 2 线程 | 4 线程 | 8 线程 | 16 线程 |
|---|---|---|---|---|---|
| 1M | 1.000 | 1.520 | 1.074 | 0.792 | 0.737 |
| 4M | 1.000 | 1.937 | 2.107 | 2.029 | 1.692 |
| 16M | 1.000 | 1.426 | 2.541 | 3.318 | 2.953 |
| 64M | 1.000 | 1.731 | 2.443 | 2.780 | 2.043 |
| 128M | 1.000 | 1.998 | 4.144 | 3.921 | 3.514 |

表 7: 并行数组求和效率

| 数组规模 | 1 线程 | 2 线程 | 4 线程 | 8 线程 | 16 线程 |
|---|---|---|---|---|---|
| 1M | 1.000 | 0.760 | 0.268 | 0.099 | 0.046 |
| 4M | 1.000 | 0.968 | 0.527 | 0.254 | 0.106 |
| 16M | 1.000 | 0.713 | 0.635 | 0.415 | 0.185 |
| 64M | 1.000 | 0.866 | 0.611 | 0.347 | 0.128 |
| 128M | 1.000 | 0.999 | 1.036 | 0.490 | 0.220 |

- **线程数量对性能的影响分析**：对于 1M 规模的数据，单线程运行时间约为 0.00409 秒，而使用 2 个线程后时间减少至约 0.00269 秒，表明在较小规模任务中，适当**增加线程**可以**降低计算时间**。然而，当线程数进一步增至 4、8 甚至 16 时，运行时间并没有保持下降趋势，相反**加速比下降**，说明线程创建、调度和同步开销开始明显占用总时间。也就是说，**多线程开销**逐渐侵蚀了并行带来的效益，这在小规模任务中尤为明显。因此，线程数量增加**并不总是**意味着更好的性能，在处理任务较小时，适量增加线程可以提高性能，但超过一定数目后，同时由于核数的限制（我的电脑只有 4 核），多线程的额外开销反而会**抵消计算加速的优势**。

- **数组规模对并行效率的影响**：随着数组规模的增大，可以看到单线程的运行时间大幅增加，从而使得在多线程环境下，每个线程能够获得更多的工作量，此时并行化的计算优势便更为明显。例如，对于 128M 这样的大规模数据，运行时间虽然较长，但多个线程共同工作时能够充分利用处理器资源，使每个线程的**计算量相对充足**，从而更好地**抵消**线程创建和同步带来的**固定开销**。反之，在 1M 或 4M 这种小规模数据下，多线程执行时线程间的管理和汇总开销可能占有较大比重，导致并行效率较低。因此，数据规模越大，任务的计算密度提高，进而**有助于提升并行效率**。

- **可扩展性分析**：当问题规模固定时，增加线程数仅能获得**有限的加速**，这体现了**有限的强可扩展性**。在并行数组求和实验中，1M 数据任务随着线程数增加到 4、8、16 时表现出加速比下降甚至**负扩展现象**，属于无可扩展；而对于 128M 数据任务，虽然增加线程数可以获得一定的加速效果，但加速比始终**远低于**理想线性扩展，说明系统受到内存带宽、同步延迟和缓存一致性等硬件资源限制，表现为弱可扩展性。总体上，实验结果显示，并行数组求和可扩展性十分有限，甚至可以认为**无可扩展性**。

- (选做) 不同聚合方式的比较：

表 8: 不同聚合方式的运行时间

| 数组规模 | 1 线程 | 2 线程 | 4 线程 | 8 线程 | 16 线程 |
|---|---|---|---|---|---|
| 1M | 0.00728 s | 0.00399 s | 0.00558 s | 0.00636 s | 0.00385 s |
| 4M | 0.01650 s | 0.01111 s | 0.00540 s | 0.00649 s | 0.00656 s |
| 16M | 0.05294 s | 0.03037 s | 0.04685 s | 0.01832 s | 0.02173 s |
| 64M | 0.23350 s | 0.12082 s | 0.09645 s | 0.07572 s | 0.12996 s |
| 128M | 0.42414 s | 0.20574 s | 0.13540 s | 0.12453 s | 0.12868 s |

最初采用的方法是让每个线程计算局部和，最后在主线程中汇总局部结果；而这里则是在每个线程计算完局部和后，通过加锁立即累加到全局变量中。从数据上看，对于 1M 和 4M 这样的小规模数据，第一种方式的运行时间明显较低：例如 1M 时，单线程分别为 0.00409 s（局部求和后汇总）对比 0.00728 s（锁聚合）；2 线程的时间分别为 0.00269 s 与 0.00399 s，4 线程则为 0.00381 s 与 0.00558 s。这表明在小规模数据下，减少频繁的锁操作和同步开销有利于降低总体耗时。而在更大规模的 16M、64M 和 128M 测试中，两种方式的运行时间差异就变得不那么显著。例如，在 16M 数据下，1 线程时分别为 0.05151 s 和 0.05294 s，相差很小；64M 数据下，1 线程分别为 0.20709 s 与 0.23350 s，而 128M 时单线程甚至略快，分别为 0.44321 s 与 0.42414 s。

表 9: 不同聚合方式的加速比

| 数组规模 | 1 线程 | 2 线程 | 4 线程 | 8 线程 | 16 线程 |
|---|---|---|---|---|---|
| 1M | 1.000 | 1.825 | 1.305 | 1.144 | 1.892 |
| 4M | 1.000 | 1.485 | 3.056 | 2.542 | 2.516 |
| 16M | 1.000 | 1.744 | 1.130 | 2.889 | 2.437 |
| 64M | 1.000 | 1.932 | 2.420 | 3.084 | 1.795 |
| 128M | 1.000 | 2.061 | 3.131 | 3.406 | 3.297 |

表 10: 不同聚合方式（锁聚合）的效率

| 数组规模 | 1 线程 | 2 线程 | 4 线程 | 8 线程 | 16 线程 |
|---|---|---|---|---|---|
| 1M | 1.000 | 0.913 | 0.326 | 0.143 | 0.118 |
| 4M | 1.000 | 0.743 | 0.764 | 0.318 | 0.157 |
| 16M | 1.000 | 0.872 | 0.283 | 0.361 | 0.152 |
| 64M | 1.000 | 0.966 | 0.605 | 0.386 | 0.112 |
| 128M | 1.000 | 1.031 | 0.783 | 0.426 | 0.206 |

对于小规模数据（如 1M），锁聚合方式的效率（0.913、0.326、0.143、0.118）明显高于局部求和后再汇总方式（分别为 0.760、0.268、0.099、0.046）。这说明在数据量较小、线程数较多时，锁聚合方式虽然引入了锁竞争，但由于总体工作量较小，其同步更新反而在某些配置下能够更充分地利用额外的线程。然而，随着数据规模增大，两种方式的效率趋于相近，并且整体效率均下降，这主要是由于线程同步和汇总开销在多线程环境下始终存在，且扩展性有限。在 4M 和 16M 测试中，局部求和方式在某些线程数（如 4 线程）下表现更好，而在 2 线程和 8 线程下锁聚合方式略占优势；在 64M 和 128M 数据中，两种方式效率相差较小，但局部汇总方式在高线程数（例如 16 线程）下略好。总体而言，对于本次实验来说，聚合方式的选择影响了小数据规模下的并行效率，而在大规模数据下，由于计算量足够，两个方法的**效率差异不大**。