南開大學
Nankai University

计算机学院

并行程序设计期末报告

倒排索引求交算法 MPI 编程

姓名：刘家骥

学号：2211437

专业：计算机科学与技术

2024 年 6 月 9 日

目录

1 问题描述

1.1 期末研究问题

倒排索引是一种通过映射来表示某个单词在一个文档或者一组文档中的存储位置的索引方法。倒排列表求交是当用户提交了一个 k 个词的查询，表求交算法返回各个倒排索引的交集。本实验拟实现倒排索引求交的串行算法，并采用并行化的方法，对算法进行优化和测试。了解并实现倒排索引求交的两种算法，先实现 list-wise 和 element-wise 两种思路的串行算法，然后从多个角度研究相应并行算法的实现，如从减少 cache 未命中次数，减少比较次数、实现指令级并行（相邻指令无依赖）以利用超标量架构等等方面进行尝试，并且比较优化的性能。

1.2 本次实验子问题及分工

1. 按表求交算法的 MPI 编程，负责人：王俊杰
2. 按元素求交算法的 MPI 编程，负责人：刘家骥

本文为刘家骥负责在 arm 和 x86 平台上的 element-wise 串行算法 MPI 并行算法的实现、与 openMP 算法的结合，以及探讨不同进程数和不同问题规模下，各算法的性能比较。

有关代码文件已经上传到Github上。

2 实验平台

2.1 硬件平台

x86 架构

- CPU：12th Gen Intel(R)Core(TM)i7-12700H
- CPU 核心数：14
- 指令集架构：x86 架构
- GPU：NVIDIA GeForce RTX 3060 Laptop
- 内存容量：16GB
- L1 cache: 1.2MB
- L2 cache: 11.5MB
- L3 cache: 24MB

ARM 架构

- 华为鲲鹏服务器
- CPU 核心数：96
- 指令集架构：aarch64
- L1d cache: 64K

- L1i cache: 64K
- L2 cache: 512K
- L3 cache: 49152K

2.2 软件平台

x86

- 操作系统及版本：Windows 11 家庭中文版
- 编译器及版本：GNU GCC Compiler
- 编译选项：Have g++ follow the C++17 GNU C++ language standard(ISO)

  Optimize even more(for speed) [-O2] 加入-fopenmp

3 倒排索引求交 MPI 编程

3.1 设计思路

由于倒排索引的查询文件中有一千条查询，而对于每个查询，其内部算法实现都是一样的，都是通过找出每个查询中的最短向量，进而与其他的向量进行求交运算。

所以我考虑了可以在每个查询间进行 MPI 编程，类似多线程进行。将数据集按照分配的进程数进行划分。每一个进程负责一部分数据集的处理，在处理之后都汇总到根进程中，并在根进程中输出结果。

如图3.2所示，主进程读取数据文件并广播到所有进程，各进程并行处理查询任务，最后汇总和输出结果。

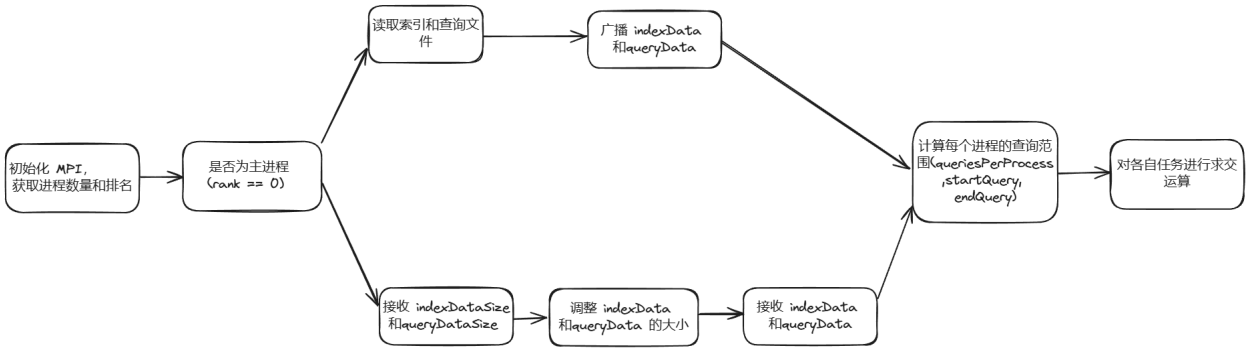


图 3.1: MPI 编程逻辑图

3.2 算法实现

x86 平台和 arm 平台上算法实现一致，同时采用 <chrono> 头文件来进行时间测量，在 arm 平台和 x86 平台上都可以用，将时间设置为以 ms 为单位。

```
1 std::chrono::duration<double, std::milli> TotalTime = end-start;
```

主进程负责读取和广播数据，所有进程（包括主进程）共同参与查询处理，并将结果收集到主进程进行汇总和输出

同时所有进程的计时结果也需要一并收集到主进程，在主进程中进行输出

Algorithm 1 MPI 并行算法

```
Initialize MPI
Get world_size and world_rank
if world_rank == 0 then
    Read index file
    Read query file
end if
Broadcast indexDataSize
Broadcast queryDataSize
if world_rank != 0 then
    Resize indexData
    Resize queryData
end if
Broadcast indexData
Broadcast queryData
Calculate queriesPerProcess
Calculate startQuery and endQuery
Start timer
for each query from startQuery to endQuery do
    Find the shortest list in the query
    Compute the intersection of lists
end for
End timer
Gather timing results
if world_rank == 0 then
    Print timing results
end if
Finalize MPI
```

3.3 实验结果

3.3.1 x86

在 x86 平台上分别测试了进程数为 2、4、6、8、16、32 的情况下，各个进程的运行时间。

进程数	运行时间/ms				
0	356333	183292	95305.7	62325.9	54661.4
1	373917	175422	90578.7	55795.1	48939.5
2	\	183262	89756.9	54357.5	32243.8
3	\	192421	86745.9	60097.3	44511.7
4	\	\	93619	53312.1	45831.5
5	\	\	97246.7	57874.4	39754.8
6	\	\	96164.2	59117.5	44142.2
7	\	\	99272.8	50716.9	41274.4
8	\	\	\	56023.8	31834.8
9	\	\	\	58903.2	45461.2
10	\	\	\	60600.1	41774.7
11	\	\	\	54467.6	39259.8
12	\	\	\	58598.8	49119
13	\	\	\	63109.6	47330.1
14	\	\	\	62216.7	25078.7
15	\	\	\	52228.5	50330.4
16	\	\	\	\	42674.8
17	\	\	\	\	44413
18	\	\	\	\	44638
19	\	\	\	\	43151.9
20	\	\	\	\	41145.2
21	\	\	\	\	38327.1
22	\	\	\	\	41927.4
23	\	\	\	\	39678.1
24	\	\	\	\	55723.2
25	\	\	\	\	43656.6
26	\	\	\	\	44422.6
27	\	\	\	\	42226.4
28	\	\	\	\	56037.9
29	\	\	\	\	46205.4
30	\	\	\	\	49958.1
31	\	\	\	\	11673
TOTAL_TIME	373917	192421	99272.8	63109.6	56037.9

我们大体上观察一下表格中的数据不难发现，在线程数为 16 和 32 时，数据之间的分布并不均匀，进程与进程之间的运行时间相差较大，存在负载不均衡的情况，在“比较分析”模块中，我将画图具体分析。

不同算法程序运行总时长如下表

进程数	串行	2	4	8	16
TOTAL_TIME/ms	730634.7	373917	192421	99272.8	63109.6

3.3.2 ARM

进程标号	运行时间/ms			
0	289697	105459	54045.3	53696.8
1	297990	100379	51491.2	49479.3
2	\	104928	50943.8	48900.2
3	\	110968	49853	51766.4
4	\	\	51518	47384.4
5	\	\	53992.1	49634.3
6	\	\	54919.1	51147.3
7	\	\	56373.5	44976.7
8	\	\	\	49188.6
9	\	\	\	51269.4
10	\	\	\	52520.1
11	\	\	\	48388.2
12	\	\	\	50798.7
13	\	\	\	54121.4
14	\	\	\	53629
15	\	\	\	46607.6
TOTAL_TIME	297990	110968	56373.5	54121.4

随着进程标号的增加，每个进程的运行时间总体上呈现下降趋势，但不稳定，有些进程的运行时间波动较大。

总体上并行化带来了运行时间的减少，表明并行化在某种程度上提升了效率。但是运行时间的减少并不是线性或稳定的，这可能是因为负载不均衡、进程间通信开销、I/O 操作瓶颈等因素造成的。

进程数	1	2	4	8	16
TOTAL_TIME/ms	421446	297990	110968	56373.5	54121.4

3.4 比较分析

通信开销：进程间的通信可能会引入额外的开销，从而影响整体性能。尤其是当进程数增加时，通信开销可能变得更加显著。

可以看出，由于查询数据有 1000 组，按照 16 和 32 划分均会出现较大波动，原因是各个进程负载不均衡，数据量不一致，导致有些进程的时间被浪费掉了，从而减弱了优化效果。

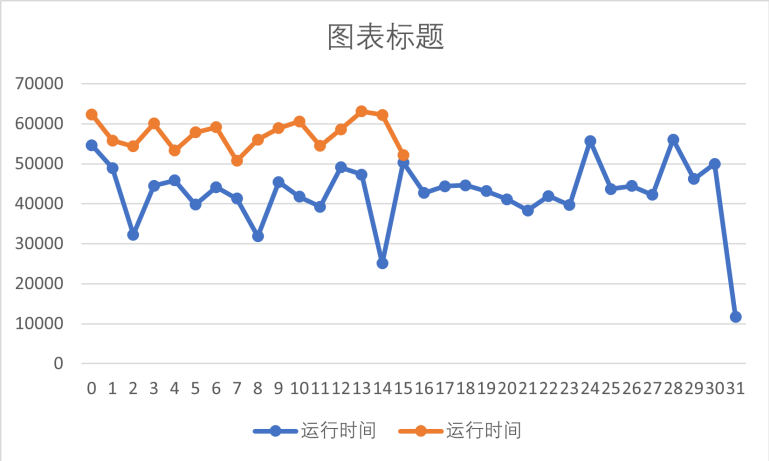
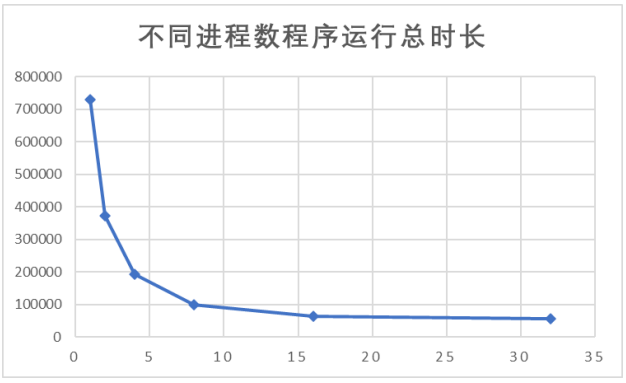


图 3.2: x86 不同进程数运行比较

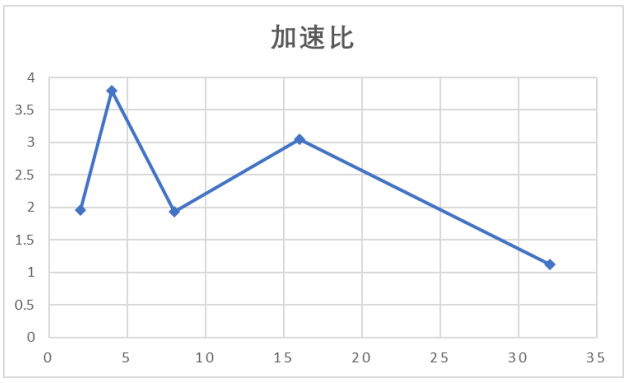
如图4(a), 可以明显看出，随着进程数增加，程序运行时间明显减少，优化效果非常好，对于只有一个进程（即串行算法），在图**??**中可以看出，两个进程的加速比几乎达到了 2，优化非常理想。

随着进程数的增多，运行总时长的下降率明显放缓，优化力度明显减弱，加速比也整体呈现减小的趋势。

究其原因，一方面应该是负载不均衡，任务不能均匀地分配到每个进程，那么一些进程可能会在等待其他进程完成工作时处于空闲状态，这会降低整体的效率。另一方面每个进程都需要与其他进程进行通信以共享数据，随着进程数的增加，通信开销也会增加。这种开销可能会抵消并行计算带来的性能提升。



(a) x86 不同进程数运行总时长比较



(b) x86 加速比

如图3.3, 在 arm 平台上进行测试，可以明显发现，当进程数超过 8 后，算法性能几乎不再有优化，运行时间在 50000ms 左右徘徊。

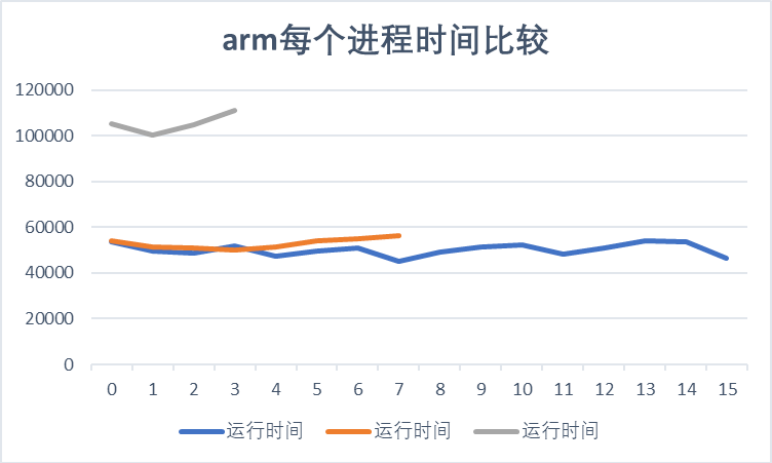
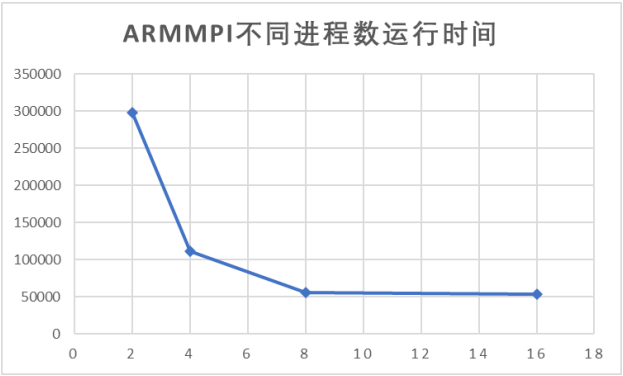
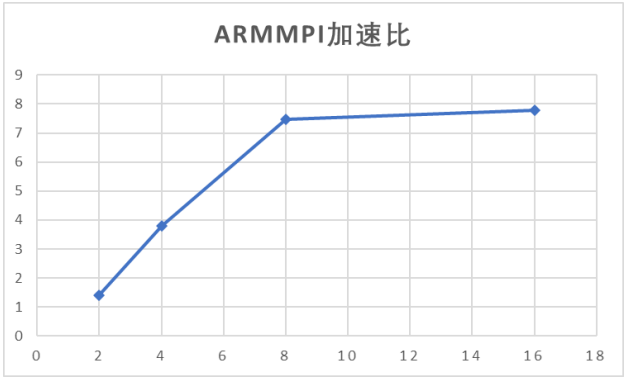


图 3.3: arm 不同进程数运行比较

可以看出，随着进程数的提升，arm 平台上算法的运行效率也在不断提升，并且从图中趋势可以看出，性能瓶颈在大于 16 个线程处。



(a) ARM 不同进程数运行总时长比较



(b) arm 加速比

4 MPI 与多线程结合

4.1 设计思路

首先本元素求交的算法并不适合进行 SIMD 化，会降低程序运行的效率，故此处仍采用串行算法，将 MPI 与 openMP 相结合。

MPI 并行已经将数据进行了分组，分为不同的进程执行，这样进程之间可以并行执行，大大提高了执行效率。

由于每个进程中间还可以进行多线程的并行，因此，程序的性能还有望继续提升。

使用 OpenMP 的并行 for 循环（#pragma omp parallel for），在每个进程内部并行处理其分配的查询。每个线程都会处理一部分查询，这部分查询是由 OpenMP 自动分配的。我们通过 omp_set_num_threads(); 来规定并行的线程数量，此处我取的值为 5，即每个进程内五个线程一起并行。

4.2 算法实现

在每个进程的查询内部，并行处理查询，使用 OpenMP 创建多线程，并行处理每个进程的查询范围内的所有查询。对于每个查询，找到最短的列表，然后计算这个列表和其他所有列表的交集。

x86 平台和 arm 平台上算法实现一致，同样采用 <chrono> 头文件来进行时间测量，将时间设置为以 ms 为单位。

```
1 std::chrono::duration<double, std::milli> TotalTime = end-start;
```

Algorithm 2 MPI+OPENMP

```
procedure ParallelQueryProcessing
    MPI_Init()
    world_size ← MPI_Comm_size()
    world_rank ← MPI_Comm_rank()
    if world_rank == 0 then
        indexData ← readIndexFile()
        broadcast(indexData)
        queryData ← readQueryFile()
        broadcast(queryData)
    end if
    queriesPerProcess ← calculateQueriesPerProcess(queryDataSize, world_size)
    startQuery, endQuery ← calculateQueryRange(world_rank, queriesPerProcess, queryDataSize)
    startTimer()
    omp_set_num_threads(5)
    for i in range(startQuery, endQuery) do
        S ← findShortestList(queryData[i])
        for all list in queryData[i] do
            S ← intersection(S, list)
        end for
        printProcessAndThreadInfo(world_rank, omp_get_thread_num())
    end for
    elapsedTime ← stopTimer()
    gatherElapsedTime(elapsedTime)
    if world_rank == 0 then
        printElapsedTime()
    end if
    MPI_Finalize()
end procedure
```

4.3 实验结果

4.3.1 x86

统计了 2、4、8、16 个进程中各个进程的运行时间，可以明显看出，随着进程数的增加，运行时间明显缩短，代码性能明显优化。

进程数	运行时间			
0	163373	101685	60495.2	59463.5
1	189544	89334	63955.6	53734.7
2	\	978466	56794.9	53342.7
3	\	107595	55876.7	56797.9
4	\	\	61562.9	49467.5
5	\	\	64935.7	51374.1
6	\	\	62389.6	57483.6
7	\	\	67986.4	47324.8
8	\	\	\	52143
9	\	\	\	53563
10	\	\	\	55479
11	\	\	\	49676
12	\	\	\	52357.3
13	\	\	\	59569.6
14	\	\	\	60036.4
15	\	\	\	49343.4
TOTAL_TIME	189544	107595	67986.4	60036.4

运行时长与只有 MPI 和串行算法进行比较，如图，可以明显看出，加入 MPI 后性能优化。

进程数	MPI 时间/ms	MPI+openMP 时间/ms	串行算法时间/ms
2	373917	189544	730634.7
4	192421	107595	\
8	99272.8	67986.4	\
16	63109.6	60036.4	\

4.3.2 ARM

进程标号	运行时间/ms			
0	60802.8	54409.6	53186.9	52932.9
1	61221.2	52182.6	52269.4	53524.3
2	\	53795	54076.9	53052
3	\	54242.4	52738.8	52822.2
4	\	\	51848.4	52742.6
5	\	\	53120.6	52941.4
6	\	\	53285.5	53106.9
7	\	\	54235.3	51169.1
8	\	\	\	53376.2
9	\	\	\	52367.1
10	\	\	\	53861.9
11	\	\	\	52911.9
12	\	\	\	52941.9
13	\	\	\	52581.7
14	\	\	\	53527.6
15	\	\	\	52035.4
TOTAL_TIME	61221.2	54409.6	54235.3	53861.9

运行总时间与只有 MPI 的比较

进程数	MPI 时间/ms	MPI+openMP 时间/ms	串行算法时间/ms
2	297990	61221.2	421446
4	110968	54409.6	\
8	56373.5	54235.3	\
16	54121.4	53861.9	\

4.4 比较分析

可以看出，也存在负载不均衡的问题，曲线波动较大，不同进程之间的运行时间存在较大差异。

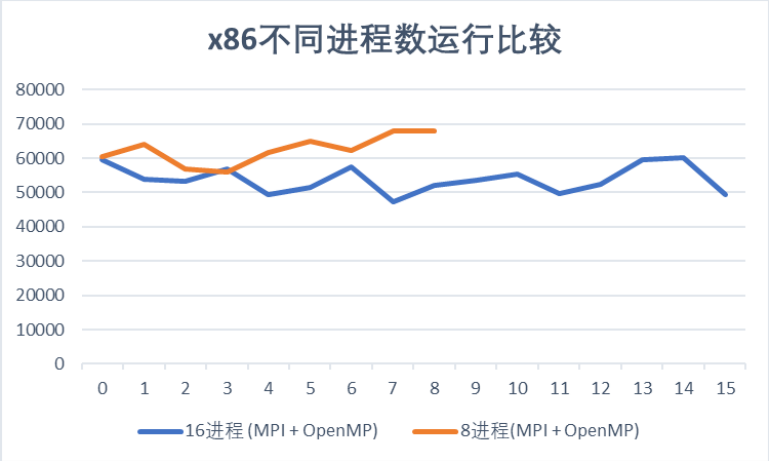


图 4.4: x86 不同进程数运行比较

如图4.6, 加入了 openmp 并行的算法性能在任何进程数上，性能都要优于只有 MPI 并行的算法，但是随着进程数的增加，二者运行时间趋于相同且趋于一个定值，即两种优化都已经接近优化极限。

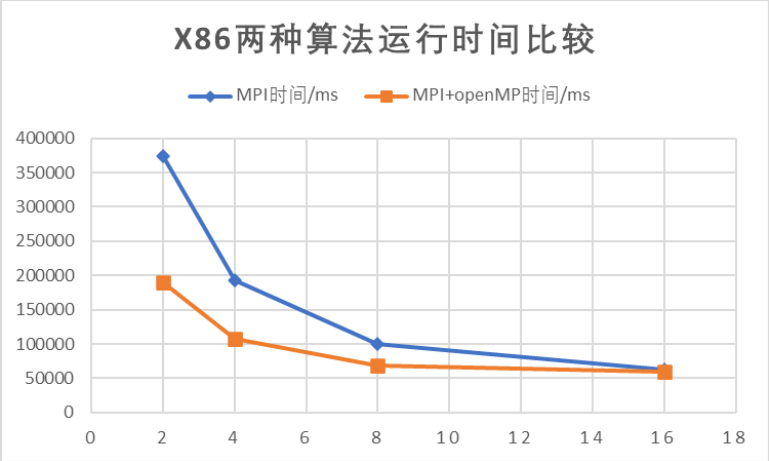


图 4.5: x86 两种算法运行比较

从加速比中可以明显看出，MPI+openMP 的组合优化力度相当之高，最高甚至能达到 12 倍左右，大大提升了程序运行的效率。

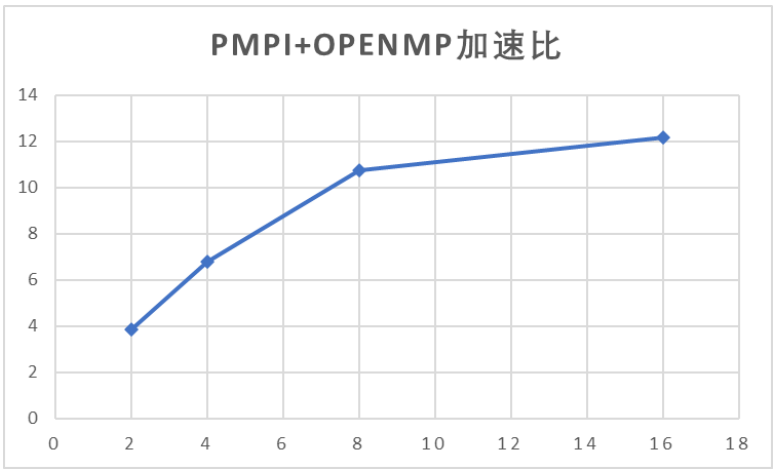


图 4.6: x86MPI+openMP 加速比

可以看出，也存在负载不均衡的问题，曲线波动较大，不同进程之间的运行时间存在较大差异。

同理在 arm 平台上我们也进行如上对比

如图4.8所示，曲线波动非常大，说明各个进程之间运行的时间相差很多，任务分配并没有特别均匀。

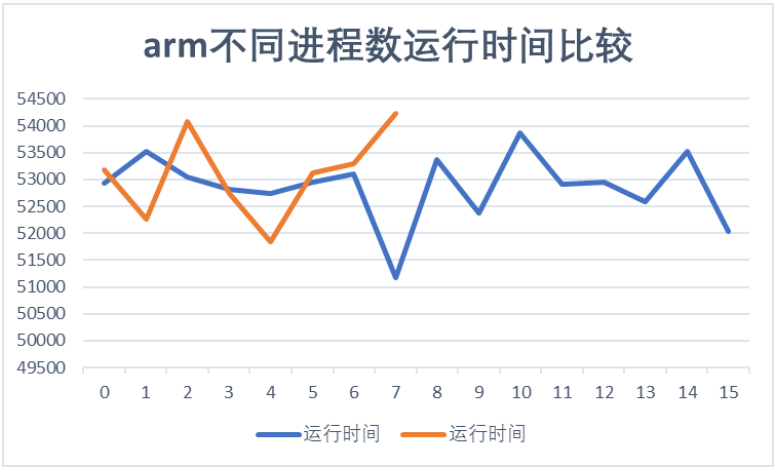


图 4.7: arm 不同进程数运行时间比较

结果很明显，加入了 openMP 算法后，程序运行效率提升很大，但是在进程数增加后，性能逐渐达到瓶颈，逐渐和只有 MPI 并行的算法相近。

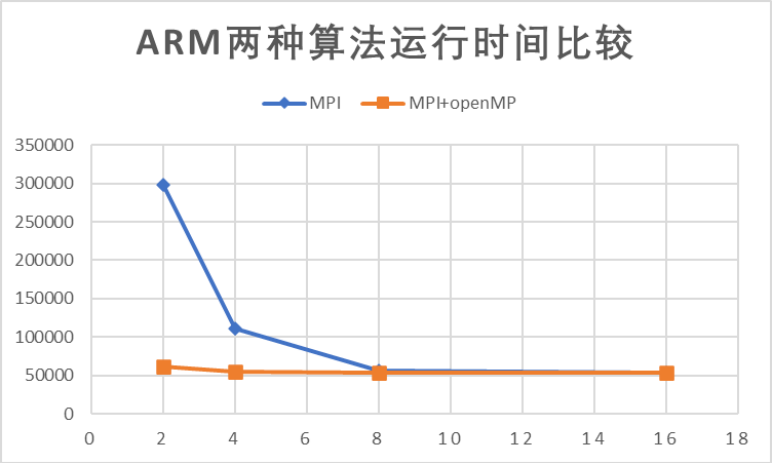


图 4.8: arm 两种算法运行时间比较

从加速比可以更加直观的看出，在超过 8 个进程后，进程数增加并不会显著增加加速比了，性能加速达到了极限。

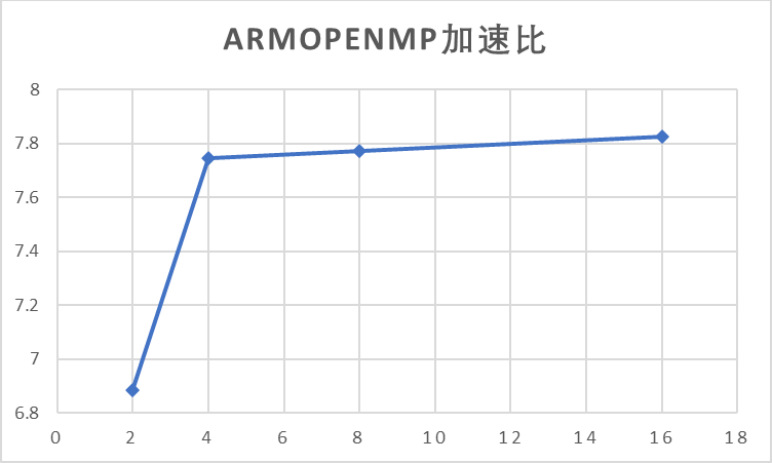


图 4.9: armopenMP 加速比

5 总结

本次实验 mpi 环境配置较为麻烦，尤其是在鲲鹏服务器上进行测试时，需要熟悉脚本的写法，每次测试需要更改脚本中的节点数。同理在 x86 平台上，需要在终端中运行 exe 程序，更改每次命令中的进程数来进行测试。

代码部分对于数据的广播和聚集稍为麻烦，同时与多线程不同，时间的测量也需要在不同进程中独立测量，在统一聚集到根进程中输出，在写代码的过程中造成不少麻烦。

通过本次实验初步掌握了 MPI 并行化的方法，并将其与 openMP 结合，极大的提升了算法的性能。还学习到了脚本的撰写方式以及 MPI 相关的环境配置，加深了对于 MPI 算法的理解。