

1 实验目的

通过实现并比较MPI（Message Passing Interface）和OpenMP（Open Multi-Processing）两种并行计算框架，加深对并行计算原理和实践的理解。具体目标为通过K-means聚类算法，掌握在集群环境下的MPI并行计算以及在单节点上的OpenMP并行计算，进一步比较它们在不同问题规模和处理器数量下的性能表现。

2 实验内容

1. **实验环境搭建**：通过Docker容器创建MPI实验环境，模拟集群环境下的并行计算。创建一个manager容器和两个node容器，并通过Docker的网络功能实现它们之间的通信，通过Docker卷来实现文件系统的共享。
2. **MPI并行计算**：在单主机上模拟MPI实验，使用Docker容器模拟多个节点。主进程作为管理节点负责数据的加载、分发、初始化聚类中心、以及最后的结果汇总。从进程负责计算和更新聚类中心。
3. **OpenMP并行计算**：实现OpenMP版本的K-means聚类算法，主要并行化数据点分类和聚类中心的更新过程。通过线程池的方式提高并行性。
4. **性能分析**：分别使用MPI和OpenMP并行计算框架进行K-means聚类，并记录不同问题规模和处理器数量下的运行时间。比较MPI和OpenMP的加速比和效率，以及不同规模和处理器数量对性能的影响。深入分析MPI和OpenMP在不同场景下的优劣势。
5. **总结与对比**：总结MPI和OpenMP在实验中的应用，分析其优势和局限性，以及在不同场景下的适用性。通过实验结果，对比两种并行计算框架的性能表现和编程难度，为选择合适的并行计算方案提供参考。

3 实验原理

K-means聚类

K-means聚类是一种常用的无监督学习算法，用于将数据集划分为K个不同的簇，每个簇内的数据点与簇内其他数据点的相似度较高，而其他簇的数据点的相似度较低。其原理可以简要概括为以下几个步骤：

1. **初始化**：随机选择K个数初始的聚类中心。
2. **分配数据**：对于每个数据点，计算其与K个聚类中心的距离，将数据点分配到距离最近的聚类中心所代表的簇中。
3. **更新**：对于每个簇，计算新的聚类中心。
4. **迭代**：重复步骤2、3，直到聚类中心不再发生显著变化，或者达到预定的迭代次数。

4 实验环境

由于条件限制，只有一台主机，所以要考虑如何充分模拟集群环境下的并行计算。我经过资料搜索，以及之前使用过docker容器的经验，为了简化配置和管理，选择使用docker来创建容器化的MPI实验。

单主机上的MPI实验

使用docker容器来模拟多个节点。创建一个manager容器和两个node容器，并通过docker的网络功能实现它们之间的通信，通过docker卷来实现文件系统的共享。

多节点通信

使用docker的网络功能创建了一个自定义网络（`mpi_network`），所有MPI容器都连接到这个网络上。这使得容器之间可以通过主机名进行通信。MPI程序可以通过指定主机名和进程数量来在这些容器之间启动。在实验中创建了一个manager容器和两个node容器，它们分别使用主机名 `manager`、`node1` 和 `node2`。

在此之前的失败尝试

尝试将本机作为manager，docker中的容器作为node，但却屡屡遭遇manager与node之间的通信问题：

1. SSH连接问题：

- 尝试通过SSH连接到容器时，遇到了主机名无法解析或连接超时的问题。
- 使用容器的 ID 进行SSH连接也未成功。













2. IP地址连接问题：

- 尝试通过容器的IP地址进行SSH连接，仍然遇到连接超时的问题。

3. 网络配置问题：

- 尽管创建了自定义网络（`mpi_network`），但无法通过主机名或IP地址进行容器之间的通信。
- 容器的网络配置和主机的网络配置可能存在不匹配。

几番尝试无果，耗费了大量时间，于是决定将manager也作为容器，将manager与node加入同一子网中，彼此使用ssh连接，共享文件系统。

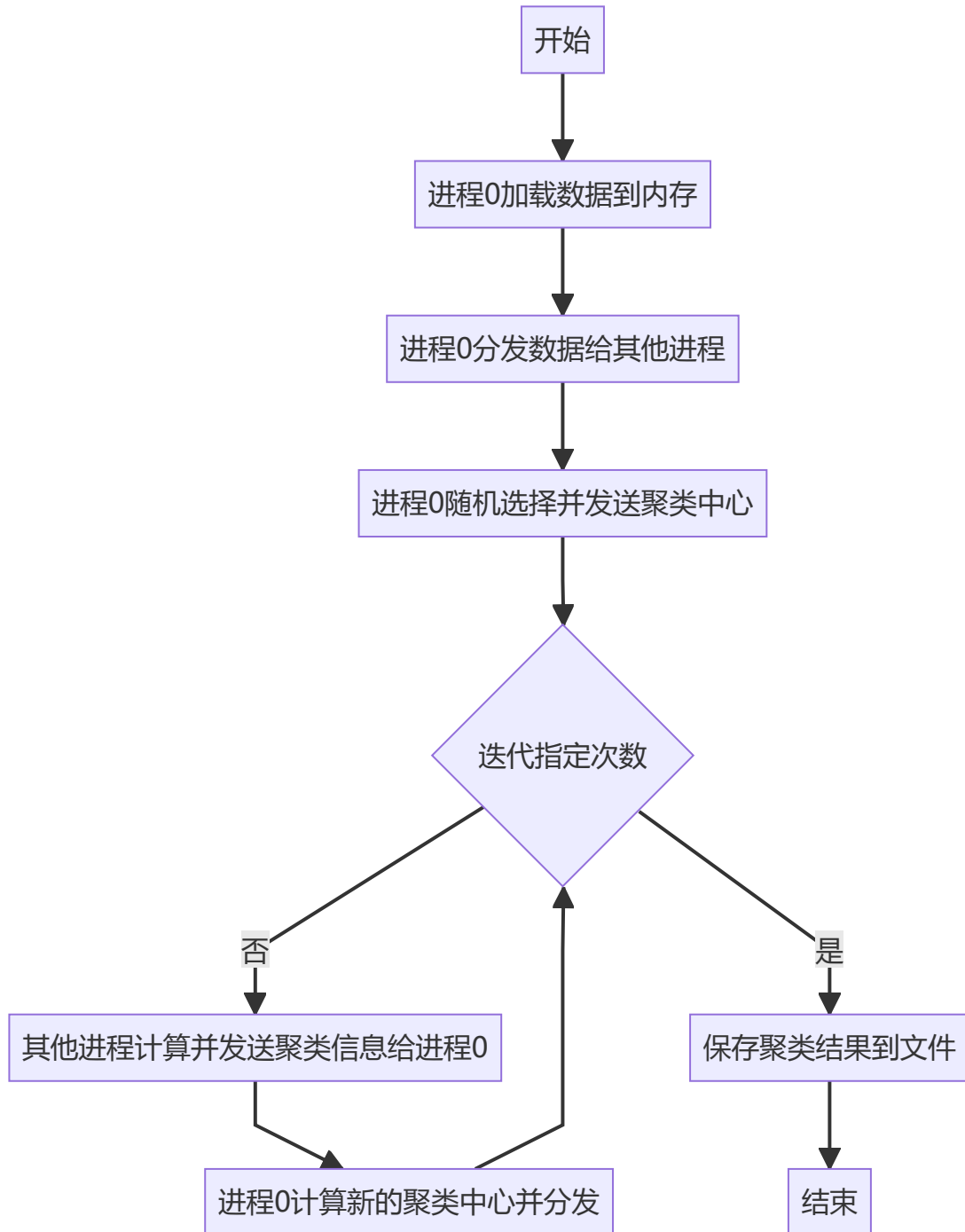
<input type="checkbox"/>	 manager fa740936513c ubuntu:latest	Running	50 minutes ago	0.73%	  
<input type="checkbox"/>	 node1 00fd3b1c9b41 ubuntu:latest	Running	50 minutes ago	0%	  
<input type="checkbox"/>	 node2 93d8fd6ce62d ubuntu:latest	Running	46 minutes ago	0%	  

5 实验设计

采用主从方式进行并行聚类：

1. **数据加载**：进程0作为主节点，从文件 `zoo.data` 中读取动物数据集，将数据按照指定的结构体 `animal` 保存到内存中。
2. **数据分发**：主节点向其他从节点分发数据，告知每个节点处理的数据量和范围。动物数据包含名字（一维），特征（十六维）、类型等属性。
3. **初始化聚类中心**：主节点随机选择每个聚类的中心点，并发送给从节点。
4. **计算距离和归类**：从节点根据分配的数据，计算每个点到各个聚类中心的距离，将点归类到距离最小的聚类。同时计算每个聚类的数据量和属性的累加和，然后进行规约。
5. **更新聚类中心**：主节点根据归类结果和属性累加和计算新的聚类中心，并发送给其他进程。
6. **迭代更新**：重复步骤4、5，直到达到指定的迭代轮数。

7. **结果保存**：将最终的聚类结果保存到文件中，每个聚类包含同一类的动物。



6 实验步骤

1. 主进程读取数据，同时告知从进程它需要处理的数据量

```
1 | if(my_rank==0){
2 |     datanum=loadData("zoo/zoo.data",data);
3 |
4 |     for(int i=1;i<comm_sz;i++){
5 |         int nums=datanum/(comm_sz-1);
6 |         int a=(i-1)*nums;
7 |         int b=i==comm_sz-1?datanum:i*nums;
8 |         int sendNum=b-a;
9 |         MPI_Send(&sendNum,1,MPI_INT,i,0,MPI_COMM_WORLD);
```

```

10     }
11     }else{
12         MPI_Recv(&datanum,1,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
13         printf("my rank:%d, my datanum:%d\n",my_rank,datanum);
14     }

1 my rank:1, my datanum:33
2 my rank:3, my datanum:35
3 my rank:2, my datanum:33

```

2. 主进程向从进程分发数据

```

1     if(my_rank==0){
2         for(int i=1;i<comm_sz;i++){
3             int nums=datanum/(comm_sz-1);
4             int a=(i-1)*nums;
5             int b=i==comm_sz-1?datanum:i*nums;
6             MPI_Send((void *) (data+a),sizeof(animal)*(b-
a),MPI_BYTE,i,0,MPI_COMM_WORLD);
7         }
8     }else{
9
10        MPI_Recv(data,sizeof(animal)*datanum,MPI_BYTE,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
11        printf("my rank:%d\n",my_rank);
12        for(int i=0;i<datanum;i++){
13            printf("%d\n",data[i].name);
14        }

1 my rank:1
2 0
3 .....
4 32
5 my rank:2
6 33
7 .....
8 65
9 my rank:3
10 66
11 .....
12 100

```

3. 主进程随机初始化聚类中心，并将初始聚类中心以广播形式发送给从进程

```

1     if(my_rank==0){
2         int visit[N];
3         memset(visit,0,sizeof(visit));
4
5         srand((unsigned int)(time(NULL)));
6         int i=0;
7         while(i<K){
8             int idx=rand()%datanum;
9             if(!visit[idx]){
10                 for(int j=0;j<D;j++){
11                     cluster_center[i][j]=data[idx].characters[j];
12                 }
13                 visit[idx]=1;

```

```

14         i++;
15     }
16 }
17 }
18
19 MPI_Bcast(cluster_center,K*D,MPI_DOUBLE,0,MPI_COMM_WORLD);

```

4. 开始循环迭代更新聚类中心，在每轮迭代中，从进程调用聚类函数计算本地数据点对应的本地聚类中心和每个聚类中心中的数据点数量，将从进程的本地结果规约到主进程。然后，主进程根据各个从进程计算后得到的信息更新聚类中心，并将新的聚类中心通过广播发送给从进程。

```

1  double local_cluster_center[K][D];
2  int local_cnt[K];
3  for(int round=0;round<epoch;round++){
4      memset(local_cluster_center,0,sizeof(local_cluster_center));
5      memset(local_cnt,0,sizeof(local_cnt));
6
7      if(rank!=0){
8          cluster(data,datanum,cluster_center,local_cluster_center,local_cnt);
9      }
10
11      memset(cluster_center,0,sizeof(cluster_center));
12      memset(total_cnt,0,sizeof(total_cnt));
13
14      MPI_Reduce(local_cluster_center,cluster_center,K*D,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD
15 );
16      MPI_Reduce(local_cnt,total_cnt,K,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
17
18      MPI_Barrier(MPI_COMM_WORLD);
19
20      if(rank==0){
21          for(int i=0;i<K;i++){
22              for(int j=0;j<D;j++){
23                  if(total_cnt[i]!=0){
24                      cluster_center[i][j]/=total_cnt[i];
25                  }
26              }
27          }
28
29          MPI_Bcast(cluster_center,K*D,MPI_DOUBLE,0,MPI_COMM_WORLD);
30      }

```

5. 迭代完成后，从进程将本地数据发送给主进程，主进程接收来自其他进程的数据，并根据类型和名称重新组织数据，最后将结果写入输出文件，包含K个簇的聚类结果，每个簇包含对应点的名称。

```

1  if(my_rank!=0){
2      int buf[datanum*2];
3      for(int i=0;i<datanum;i++){
4          buf[i*2]=data[i].name;
5          buf[i*2+1]=data[i].type;
6      }
7      MPI_Send(buf,datanum*2,MPI_INT,0,0,MPI_COMM_WORLD);
8  }else{
9      int buf[datanum*2];

```

```

10     for(int i=1;i<comm_sz;i++){
11         int nums=datanum/(comm_sz-1);
12         int a=(i-1)*nums;
13         int b=i==comm_sz-1?datanum:i*nums;
14         MPI_Recv((void *)&buf[a*2]),(b-
a)*2,MPI_INT,i,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
15     }
16
17     int cluster[K][N];
18     int clusternum[N];
19     memset(clusternum,0,sizeof(clusternum));
20     for(int i=0;i<datanum;i++){
21         int name=buf[i*2];
22         int type=buf[i*2+1];
23         cluster[type][clusternum[type]++]=name;
24     }
25
26     FILE *output=fopen("kmeans_result_mpi.txt","w");
27     for(int i=0;i<K;i++){
28         fprintf(output,"cluster %d:\n",i);
29         for(int j=0;j<clusternum[i];j++){
30             fprintf(output,"%s\n",idx2name[cluster[i][j]]);
31         }
32     }
33     fclose(output);
34 }

```

```

1 cluster 0:
2 antelope
3 buffalo
4 .....
5 cluster 1:
6 clam
7 pitviper
8 .....
9 cluster 2:
10 armadillo
11 bear
12 .....
13 cluster 3:
14 cavy
15 hamster
16 .....
17 cluster 4:
18 crab
19 crayfish
20 .....
21 cluster 5:
22 chicken
23 crow
24 .....
25 cluster 6:
26 bass
27 carp
28 .....

```

7 实验分析

7.1 MPI性能分析

p	1	2	3	4	5	6	7	8
T(second)	26.37	31.60	20.65	18.78	16.67	14.54	13.28	11.74
S	1	0.83	1.28	1.40	1.58	1.81	1.96	2.25
E=S/p	1	0.42	0.43	0.35	0.32	0.30	0.28	0.28

注：该MPI程序采用主从方式进行并行，进程数须大于1，因此进程数等于1的相关结果是在串行程序下测试得到。

现象1：进程数为2时，加速比小于1，相较于未采用任何并行方式的串行程序，性能不升反降。

分析：当进程数为2时，主进程负责分发和收集数据，只有一个从进程在做运算。串行程序同样是一个进程在做运算，且少了进程之间通信的消耗。因此该并行程序的进程数为2时，性能不升反降。

现象2：随着进程数的增加，并行程序的加速比提升较小，效率越来越低

分析：如果用 $T_{\text{开销}}$ 表示并行开销，那么

$$T_{\text{并行}} = \frac{T_{\text{串行}}}{p} + T_{\text{开销}}$$

由于该程序中的问题规模不大，所以用于协调各进程的工作所需要的时间相对较多，因此，并行化后的性能提升不太理想。

为验证该分析，测试不同规模下并行程序的加速比和效率：

一半规模

p	1	2	3	4	5	6	7	8
S	1	0.84	1.24	1.34	1.52	1.64	1.67	1.70
E=S/p	1	0.42	0.41	0.34	0.30	0.27	0.24	0.21

原始规模

p	1	2	3	4	5	6	7	8
S	1	0.83	1.28	1.40	1.58	1.81	1.96	2.25
E=S/p	1	0.42	0.43	0.35	0.32	0.30	0.28	0.28

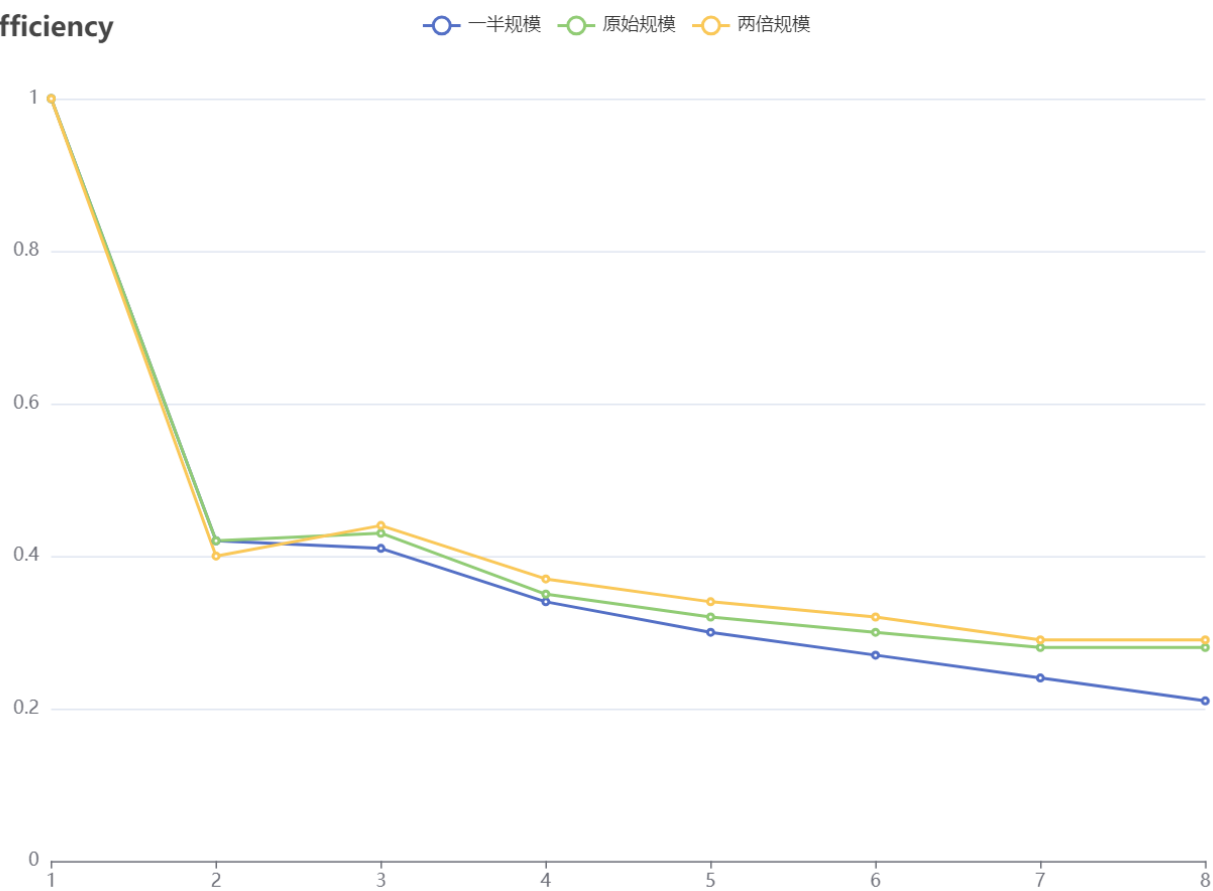
两倍规模

p	1	2	3	4	5	6	7	8
S	1	0.79	1.33	1.48	1.68	1.91	2.01	2.33
E=S/p	1	0.40	0.44	0.37	0.34	0.32	0.29	0.29

Speedup



Efficiency



现象3：当问题的规模变大时，加速比和效率增加；当问题的规模变小时，加速比和效率降低。

分析：当问题规模变大时，程序可并行的部分的比例在增加，不可并行的部分的比例在降低，以及通信和同步带来的开销相对减小，因此问题规模的增大有助于提高并行计算的性能。然而，小规模问题可并行的部分相对较少，更容易受到串行部分以及并行化开销（协调与同步）的影响，导致加速比和效率相对较低

7.2 OMP性能分析

为了研究不同并行方式的差异，我另外实现了OpenMP版本的Kmeans聚类程序，主要并行化的部分有：

- `cluster` 函数中数据点的分类过程，每个线程处理数据点集的一部分，以提高分类速度。

```
1  #pragma omp parallel for
2  for(int i=0;i<dataSize;i++){
3      // ...
4  }
```

- `main` 函数中聚类中心的更新过程。

```
1  #pragma omp parallel for
2  for(int i=0;i<K*D;i++){
3      if(total_cnt[i/D]!=0){
4          cluster_center[i/D][i%D]/=total_cnt[i/D];
5      }
6  }
```

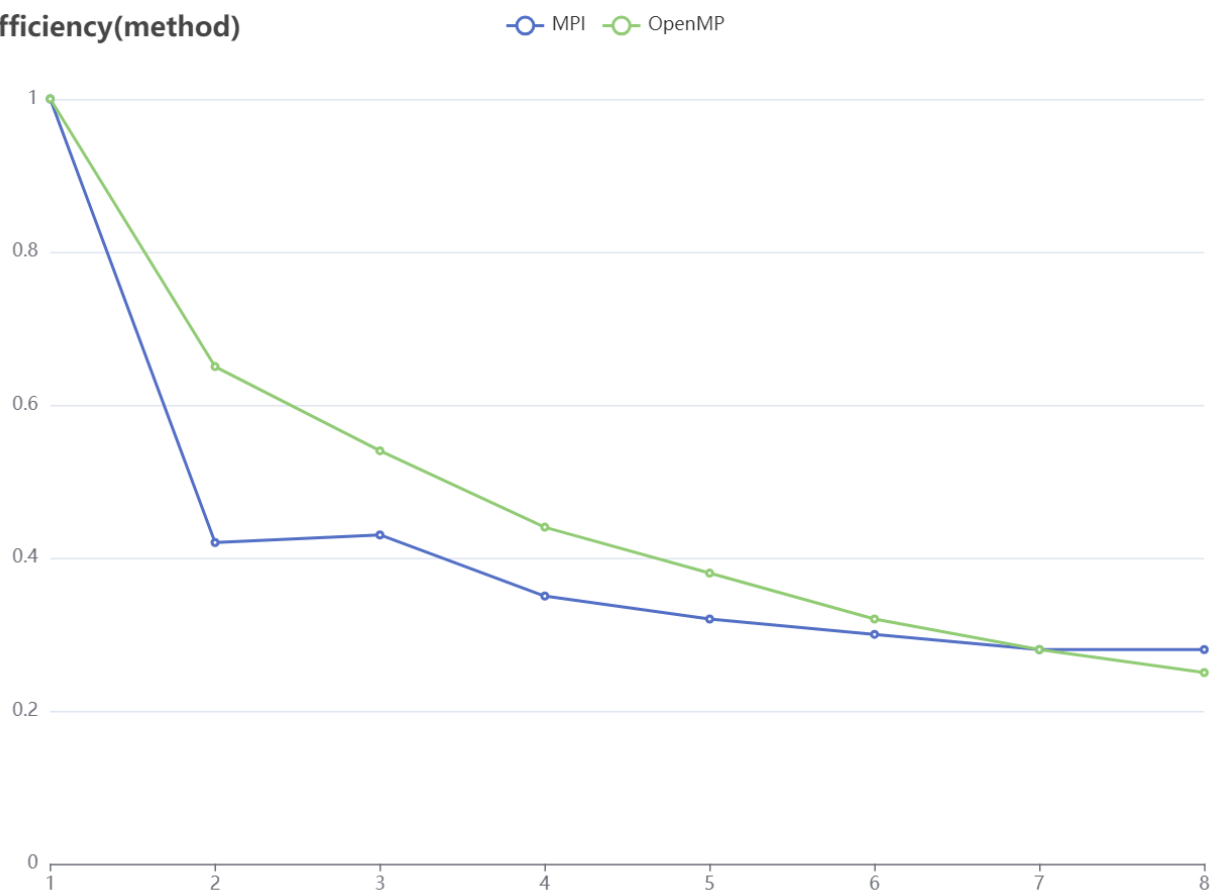
OpenMP的加速比和效率

p	1	2	3	4	5	6	7	8
T(second)	26.37	20.24	16.25	15.12	14.03	13.85	13.46	13.03
S	1	1.30	1.62	1.74	1.88	1.90	1.95	2.02
E=S/p	1	0.65	0.54	0.44	0.38	0.32	0.28	0.25

Speedup(method)



Efficiency(method)



现象1：当线程/进程数量较少时，OpenMP的性能优于MPI

分析： 由于该问题的规模不算很大，OpenMP采用共享内存机制，少了许多数据传输的开销，而MPI采用分布式内存机制，相较之下多了许多数据传输以及通信同步的开销，因此当处理器数量较少时，OpenMP的性能优于MPI。

现象2：当处理器数量超过7时，OpenMP的性能劣于MPI

分析：

1. 在共享内存系统中，过多的线程竞争可能会导致锁的争用，降低程序的性能。
2. 在共享内存系统中，内存带宽是有限的。当处理器数量超过一定阈值时，内存带宽可能成为性能的瓶颈，导致效率降低。

8 实验总结

	MPI	OpenMP
优势	适用于在不同节点间进行通信的分布式内存环境，适用于大规模集群	适用于共享内存系统，更方便地在同一节点上进行运算；程序编写相对简单，容易实现并行化
缺陷	对小/中规模的问题而言，通信开销过大；程序编写相对较复杂，需要处理进程间的通信和同步	无法处理跨节点的并行化计算，对于大规模的分布式计算集群，其可扩展性受到限制

- 在小规模数据和较少的进程/线程时，OpenMP表现更好，可能因为共享内存的优势和更简单的编程模型。
- MPI适用于大规模、分布式的计算环境，而OpenMP更适用于中小规模、单节点或共享内存的计算环境。

9 参考文献

- 奎因 (Quinn, M. J.). (2004). MPI与 OpenMP 并行程序设计: C语言版. 陈文光，武永为等译. 北京: 清华大学出版社.
- MPI Tutorial. (n.d.). Retrieved from <https://mpitutorial.com/tutorials/>