

石邢越 16337208

张永东教授

高性能计算程序设计基础

2018 年 10 月 9 日

## 实验 4: 用 MPI 实现稀疏矩阵-向量乘法的并行算法

### 1 实验目的

在“实验 3: 用 MPI 实现稠密矩阵-向量乘法的并行算法”的基础上, 探索可以更好地适应稀疏矩阵-向量乘法的并行算法。

与实验 3 的主要不同点为:

- 矩阵  $A$  为稀疏矩阵。在分发、计算时, 为避免不必要的通信、计算, 可以不使用完整的矩阵  $A$ , 而使用去除了大部分空元素得到稠密矩阵  $A'$ 。

则首先需要实现得到  $A'$  的算法。

### 2 实验要求

根据非零元分布划分矩阵: 每个进程有矩阵的  $N/p$  个非零元,  $N$  为非零元个数,  $p$  为进程数;

每个进程含有全部的向量元素。

将矩阵重复成倍扩展, 验证算法的可扩展性

在实现并行的稀疏矩阵-向量乘法时, 尝试分别使用以下 4 种方法来将矩阵  $A'$  中的元素分发给各个计算节点, 并计算各种算法对应的  $T_p$  (并行用时),  $S$  (加速比),  $E$  (效率):

- A1: 单发;
- A2: 流水线;
- A3: 集合散发;
- A4: 并行读取对应的矩阵元素。

### 3 算法原理

#### 3.1 4种数据分发算法的并行性能分析

对于 A1, A2, A3, A4 四种分发数据的算法，设  $\mathbf{A}'$  中有  $n$  个需要发送到其他计算节点的数据，则可以算出各种算法的并行性能如图 1。之后，我将实现分别各种数据分发算法的矩阵向量乘法，观察实际实验结果与图 1 中的理论值是否一致。

Algorithm	A1	A2	A3	A4
$p$	$n^2$	$\log n$	$n$	$\sqrt{n}$
$T_P$	1	$n$	$\sqrt{n}$	$\sqrt{n} \log n$
$S$	$n \log n$	$\log n$	$\sqrt{n} \log n$	$\sqrt{n}$
$E$	$\frac{\log n}{n}$	1	$\frac{\log n}{\sqrt{n}}$	1
$pT_P$	$n^2$	$n \log n$	$n^{1.5}$	$n \log n$

图 1. 四种数据分发算法的并行性能对比

#### 3.2 得到稠密矩阵 $\mathbf{A}'$

问题中的系数矩阵  $\mathbf{A}$  大小为  $60222 * 60222$ 。考虑内存开销，将数据存在一个  $60222 * 60222$  的二维数组中会产生大量的浪费（因为零元可以不需要存到内存中，直接丢弃即可），所以我设计了数据结构 `element` 来存放稀疏矩阵  $\mathbf{A}$  中的非零元（见清单.1）。为了是这个数据结构可以在之后的 MPI 函数中使用，需要将它注册为新的 MPI 数据类型（见清单.2）。

```

1  struct element {
2      int row;
3      int col;
4      double value;
5      element(int r, int c, int v) {
6          row = r; col = c; value = v;
7      }
8  };

```

清单 1. 用于存储稀疏矩阵  $\mathbf{A}$  中非零元的数据结构 `element`

```

1  //register new MPI data structure "element"
2  int blockLength[]={1,1,1};
3  MPI_Datatype oldTypes[]={MPI_INT,MPI_INT,MPI_DOUBLE};
4  MPI_Aint addressOffsets[]={0,sizeof(int),sizeof(int),sizeof(double)};
5  MPI_Datatype newType=MPI_Datatype::Create_struct(sizeof(blockLength)/sizeof(int),blockLength,addressOffsets,oldTypes);
6  newType.Commit();

```

## 清单 2. 注册新的 MPI 数据类型

### 3.3 4 种数据分发算法

#### 3.3.1 A1: 进程 0 用 MPI\_Send 将矩阵 A 各部分发给其他进程

我先在进程 0 中声明了  $(comm\_sz - 1)$  个 element 类型数组，用来暂时存储其他进程的 local\_matrix，进程 0 发送完各个 local\_matrix 后即收回这些数组的空间。

进程 0 将矩阵 A 的各个部分放到  $(comm\_sz - 1)$  个 element 类型数组中，之后将它们都用 MPI\_Send 发送；其他进程用 MPI\_Recv 接收这些数组，放到自己的 local\_matrix 数组中。

#### 3.3.2 A2: 进程 0 用 pthread 多线程读并分发矩阵 A

A2 是对 A1 的改进。将进程 0 中读矩阵 A 和发送矩阵各部分的过程进一步用 pthread 库并行化。

我选择只对发送矩阵各部分的这里进行并行，原因是理论分析可知对读矩阵 A 并行化对减少  $T_p$  帮助甚微。具体地，如果我希望进程 0 中各个线程可以尽量并发地发送，那么发送前需要确保所有要发送给其他进程的数据都已经读取完毕。假设我在读取的过程中也使用了多线程，那么我需要等待用时最长的那个线程读取完毕才可以开始多线程发送，而这个用时最长的线程用时和单线程读取完毕整个矩阵 A 是相同的（因为如果一个线程要读文件最尾部的那些数据，它需要把前面的所有数据先读了），这表明对读矩阵 A 做多线程并行作用不大。

#### 3.3.3 A3: 用 MPI\_Scatterv 函数集合散发

MPI\_Scatter 函数存在一定限制，即每个进程必须发送正好相同数量的数据，这意味着如果进程数  $comm\_sz$  不能整除数据量  $n$  时 MPI\_Scatter 函数会出错。我设计了函数 Init\_counts\_displs 计算 MPI\_Scatter 函数需要发送给某个进程的数据量，并改用可以发送不同数据量的 MPI\_Scatterv 函数。

#### 3.3.4 A4: 各进程并发读取矩阵 A 对应部分

实验要求“所有进程同时从矩阵文件中并行读取对应的矩阵元素”，且又要求“根据非零元分布划分矩阵”，那么我需要先让一个进程（比如进程 0）了解 A 中的非零元的分布，之后才能让 n 个进程均匀地读取大约  $\frac{n}{p}$  个非零元。

从原理上看，A4 和 A3 都需要进程 0 读一遍矩阵 A 得到非零元，之后 A3 中进程 0 把非零元分发给各个进程，A4 中进程 0 根据非零元的分布告诉各个进程要读哪部分的矩阵 A，其他进程根据指示去读文件对应部分（此时进程 0 也可以不用再读一次，直接使用已经读到的矩阵 A 数据）。

那么可以粗略估计，A3 中处理矩阵 A 用时：

$$t_{\text{读取一次矩阵 A}} + t_{\text{传递一次矩阵 A}};$$

A4 中处理矩阵 A 用时：

$$\begin{aligned} & t_{\text{读取一次矩阵 A}} + t_{\text{传递一次非零元分布信息}} + \left( \frac{1}{n} * \frac{n(n+1)}{2} \right) * t_{\text{读取一次矩阵 A}} \\ &= \frac{n+3}{2} * t_{\text{读取一次矩阵 A}} + t_{\text{传递一次非零元分布信息}}; \end{aligned}$$

需要注意的是，虽然是所有进程（可能除了进程 0）分别读取自己对应的部分非零元，但是由于读文件还是只能从文件开头开始读，所有文件内容会被多次重复读取，所有上式中所有进程重新分别读矩阵 A 的时候，实际上读的数据量高达  $\left( \frac{1}{n} * \frac{n(n+1)}{2} \right) * t_{\text{读取一次矩阵 A}} = \frac{n+1}{2} * t_{\text{读取一次矩阵 A}} = O(n)$ ①。基于 A3 的结果，我有理由相信这种方法的效果也不会非常理想。

实际实验时，我对上面的方法做了一点改进：

- 进程 0 读完一次矩阵 A，将非零元存储在另外一个文件中，记为矩阵 A'，矩阵大小记为 n'，这样①式中的 n 减小为 n'，可以减少重复读取的次数。
- 非零元的信息作为公有变量，进程 0 读完一次矩阵 A 统一赋值，从而以空间换时间地减少了传递它的时间。

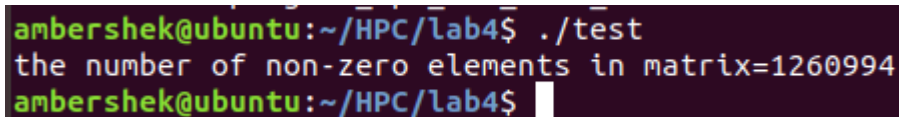
## 4 实验过程

### 4.1 每个进程得到向量 $x$

因为实验要求中要求每个进程都得到完成的向量  $x$ ，所以我使用的方法是让各个进程并发地分别读取  $x$  向量。这种算法比“由 0 号进程读入向量  $x$  后广播”使用了更少的通信量。

### 4.2 取出稀疏矩阵 $A$ 中的非零元

读取系数矩阵  $A$  的 `mtx` 文件时，首先判断读入的元素值是否为 0。只有当元素的值不为 0 时，才把这个元素存放到内存。稀疏矩阵中共有 1,260,994 个非零元（见图.2）。



```
ambershek@ubuntu:~/HPC/lab4$ ./test
the number of non-zero elements in matrix=1260994
ambershek@ubuntu:~/HPC/lab4$
```

图.2 矩阵  $A$  中共有 1,260,994 个非零元

### 4.3 各进程得到 $A$ 中的非零元

#### 4.3.1 A1: 单发

首先在进程 0 中得到进程 0 的 `local_matrix`，将其他进程的 `local_matrix` 的数据先暂时放在 `matrix[i]` 中。

```

1  while (!mat.eof()) {
2      mat >> row>>col>>value;
3      //如果是非零元
4      if (row > 0 && col > 0 && value != 0 && mat.is_open()) {
5          element new_element=element(row,col,value);
6          if (index<counts[0]){
7              //进程0的local_matrix直接得到
8              local_matrix[index]=new_element;
9          }
10         else{
11             for (int i=1; i<comm_sz; i++){
12                 //其他进程的loacl_matrix先暂时放在matrix[i]中
13                 //之后用MPI_Send发送过去
14                 if (displs[i]<index && index<displs[i]+counts[i]){
15                     matrix[i][index-displs[i]]=new_element;
16                     break;
17                 }
18             }
19         }
20         index++;
21     }
22 }

```

之后在进程 0 中将各个 matrix[i]发送出去。

```

1  for (int i=1; i<comm_sz; i++){
2      MPI_Send(matrix[i], counts[i], newType, i, 0, comm);
3      free(matrix[i]);
4  }

```

其他进程接收 matrix[i]赋值给自己的 local\_matrix。

```

1  MPI_Recv(local_matrix, local_n, newType, 0, 0, comm,
MPI_STATUS_IGNORE);

```

### 4.3.2 A2: 流水线

发送 matrix[i]和发送后的释放内存由进程 0 中的(comm<sub>sz</sub> - 1)个线程并行执行，线程调用的函数如下。

```

1  void * Send_and_free(void* rank){
2      long my_rank=(long) rank;
3      int blockLength[]={1,1,1};
4      MPI::Datatype oldTypes[]={MPI_INT,MPI_INT,MPI_DOUBLE};
5      MPI::Aint addressOffsets[]={0,sizeof(int),sizeof(int)*2};
6      MPI::Datatype newType=MPI::Datatype::Create_struct(sizeof(blockLength)/sizeof(int),blockLength,addressOffsets,oldTypes);
7      newType.Commit();
8      cout<<"Thread #"<<my_rank<<" in Process #0"<<endl;
9      MPI_Send(matrix[my_rank], counts[my_rank], newType, my_rank, 0, comm);
10     free(matrix[my_rank]);
11     newType.Free();
12     return NULL;
13 }

```

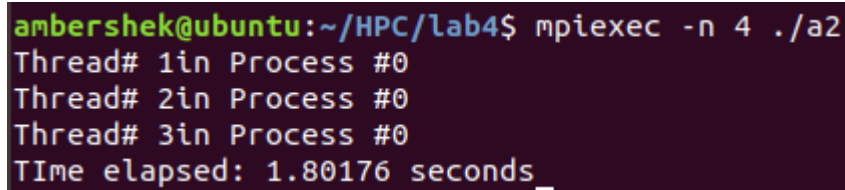
进程 0 中启动和结束线程的代码如下。

```

1 //启动线程
2 for (thread=0; thread<thread_count; thread++){
3     pthread_create(&thread_handles[thread],NULL,Send_and_free,(void*)(thread+1));
4 }
5 //结束线程
6 for (thread=0; thread<thread_count; thread++){
7     pthread_join(thread_handles[thread],NULL);
8 }
9 free(thread_handles);

```

由以下输出可以确认多个线程在正常工作。



```

ambershek@ubuntu:~/HPC/lab4$ mpiexec -n 4 ./a2
Thread# 1in Process #0
Thread# 2in Process #0
Thread# 3in Process #0
Time elapsed: 1.80176 seconds

```

图 3. 确认多个线程成功启动

### 4.3.3 A3: 集合通信 (MPI\_Scatterv 函数)

#### 4.3.3.1 第一步：计算每个进程需要分发多少个数据

调用 `Init_counts_displs` 函数计算每个进程需要从进程 0 接收的数据量和偏移量，分别存在数组 `counts` 和 `displs`。

```

1 //计算scatter函数分别要向各个进程发送多少数据量(counts), 从哪里开始发(displs)
2 void Init_counts_displs(int counts[], int displs[], int n) {
3     int offset, q, quotient, remainder;
4
5     quotient = n/comm_sz;
6     remainder = n % comm_sz;
7     offset = 0;
8     for (q = 0; q < comm_sz; q++) {
9         if (q < remainder)
10             counts[q] = quotient+1;
11         else
12             counts[q] = quotient;
13         displs[q] = offset;
14         offset += counts[q];
15     }
16 }

```

#### 4.3.3.2 第二步：使用 MPI\_Scatterv 指定数据量地分发数据

上一步得到 `counts` 和 `displs` 后，将这两个数组作为参数在 `MPI_Scatterv` 函数中使用，指定数量地向各个进程分发 `A` 中的非零元。

```
1 MPI_Scatterv(matrix, counts, displs, newType, local_matrix, local_n, newType, 0, comm);
```

### 4.3.4 A4: 并发读取

#### 4.3.4.1 第一步：非零元写入另一文件，各进程计算要读哪部分

```
1  if (my_rank==0) {
2      //读取矩阵A
3      //并将矩阵A中的非零元写入另一个文件
4      ifstream mat("/home/ambershek/HPC/lab4/matrix.mtx");
5      ofstream mat_nonzero("/home/ambershek/HPC/lab4/matrix_nonzero.mtx");
6      int row, col;
7      double value;
8      string header;
9      int index_matrix=0;
10     getline(mat, header); //delete the header
11     while (!mat.eof()) {
12         mat >> row >> col >> value;
13         if (row > 0 && col > 0 && value != 0 && mat_nonzero.is_open()) {
14             //非零元写入matrix_nonzero文件
15             mat_nonzero << row << " " << col << " " << value << "\n";
16         }
17         mat.get();
18         if (mat.peek() == EOF) break;
19     }
20 }
21
22 //给各进程共有的变量counts, displs赋值
23 //counts记录某进程要多少个非零元
24 //displs记录某进程读非零元的偏移量
25 counts=(int*)malloc(sizeof(int)*comm_sz);
26 displs=(int*)malloc(sizeof(int)*comm_sz);
27 if (displs!=NULL&&counts!=NULL) {
28     Init_counts_displs(counts, displs, n);
29 }
```

1	60222	60222	1622121	1	1 1 402108
2	1	1	402107.764651751	2	2 1 1.52122e-09
3	2	1	1.521219367046278E-009	3	3 1 -147759
4	3	1	-147758.964143426	4	4 1 1.45084e-09
5	4	1	1.450843106734568E-009	5	5 1 -7.03763e-11
6	5	1	-7.037626031172235E-011	6	6 1 -4.13711e-08
7	6	1	-4.137109499424696E-008	7	19 1 147759
8	19	1	147758.964143426	8	20 1 4.10073e-08
9	20	1	4.100729711353779E-008	9	21 1 -1.48233e-09
10	21	1	-1.482333513441292E-009	10	22 1 6.59932e-11
11	22	1	6.599320502197874E-011	11	23 1 1.46028e-09
12	23	1	1.460275461364331E-009	12	24 1 34671.8
13	24	1	34671.8020774047	13	296 1 61607.1
14	295	1	0.000000000000000E+000	14	298 1 -5951.97
15	296	1	61607.1428571235	15	299 1 -17629.5
16	297	1	0.000000000000000E+000	16	1453 1 1.46614e-09
17	298	1	-5951.97246727365		
18	299	1	-17629.4820717131		
19	300	1	0.000000000000000E+000		
20	1453	1	1.466140149723646E-009		

图.4 稀疏矩阵 A(左)和只保留非零元的稠密矩阵A'(右)。可以看到值为0的元素被丢

弃，最后A'中有 1,260,994 个非零元。

#### 4.3.4.2 第二步：各进程并发读取非零元对应部分



各进程根据上一步得到的 counts 和 displs 数组读取相应部分的非零元。

```

1      ifstream mat_nonzero("/home/ambershek/HPC/lab4/matrix_nonzero.mtx");
2      if (mat_nonzero.is_open()){
3          int row,col;
4          double value;
5          for (int i=0; i<displs[my_rank]; i++){
6              mat_nonzero>>row>>col>>value;
7          }
8          for (int i=0; i<local_n; i++){
9              mat_nonzero>>row>>col>>value;
10             element new_element=element(row,col,value);
11             local_matrix[i]=new_element;
12         }
13     }

```

#### 4.4 各进程并行进行乘法

对各进程 local\_matrix 中的每个元素，将它的 value 值乘到该列对应行的向量 x 的元素上，即

$$x_{col} = x_{col} * A_{row,col}.$$

代码实现为：

```

1  for (int i=0; i<local_n; i++){
2      x[local_matrix[i].col] *= local_matrix[i].value;
3  }

```

#### 4.5 从各进程收集数据得到最终结果并输出

将上一步处理过的各进程中的向量 **x** 用 MPI\_Reduce 求和（数组和，求和结果也是一个数组，记为 **y**），代码实现如下：

```

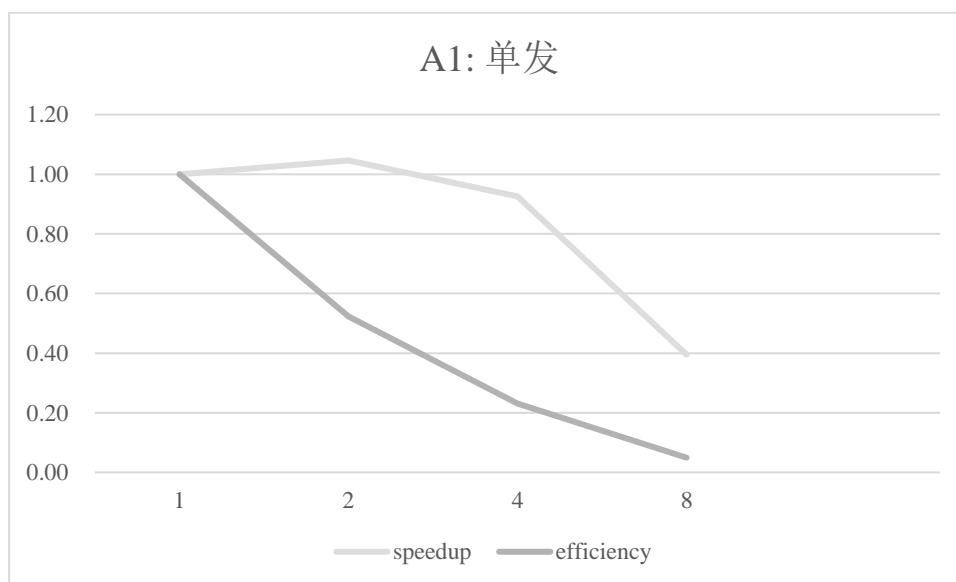
1  y=(double*)malloc(sizeof(double)*rank);
2  MPI_Reduce(&x[0], &y[0], rank, MPI_DOUBLE, MPI_SUM, 0, comm);

```

### 5 结果分析与问题讨论

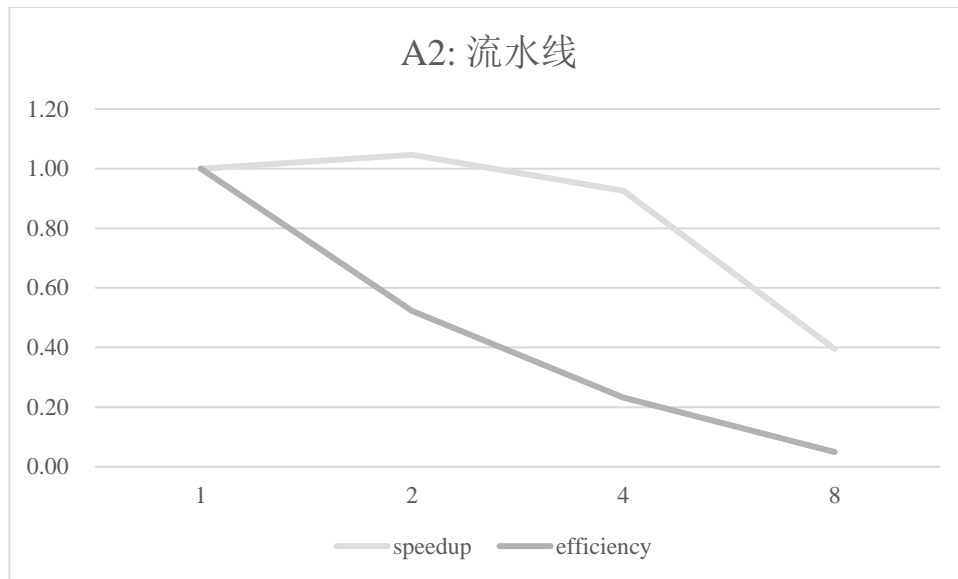
## 5.1 A1 结果

进程数	1	2	4	8	16
$T_p$ (并行用时)	1.38915	1.34055	1.56577	3.79429	9.55486
S(加速比)	1.00	1.04	0.89	0.37	0.15
E(效率)	1.00	0.52	0.22	0.05	0.01



## 5.2 A2 结果

进程数	1	2	4	8
$T_p$ (并行用时)	1.47788	1.41252	1.59614	3.74105
S(加速比)	1.00	1.05	0.93	0.40
E(效率)	1.00	0.52	0.23	0.05

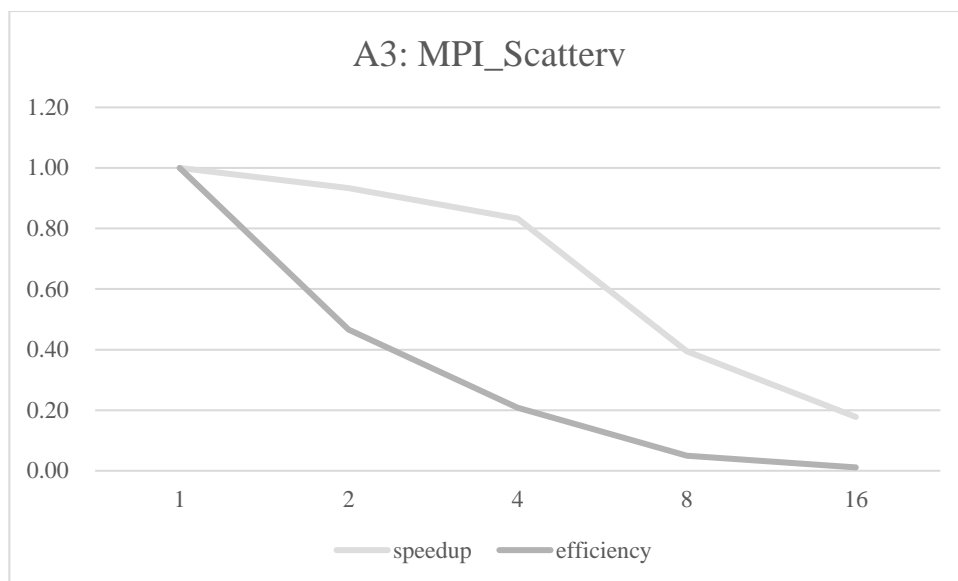


### 5.3 A3 结果

这个结果可以说是很搞笑了，并行用时比串行更长，且随进程数增加而增加。

这可能是因为计算数据量不够大，导致使用 **MPI** 后的通信用时在总用时中比例比计算用时更大，且当进程数增加时，通信用时增加得更明显。

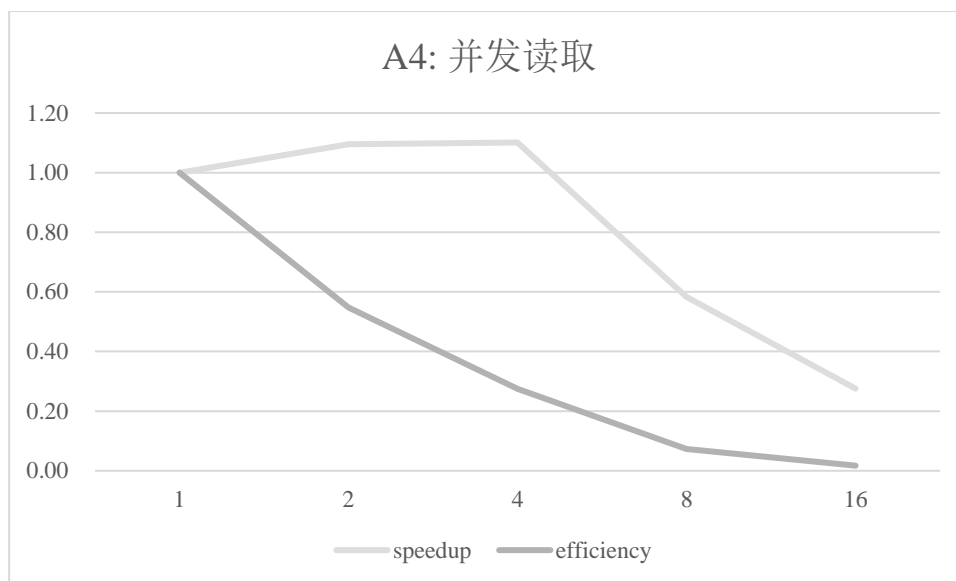
进程数	1	2	4	8	16
<b>T<sub>p</sub>(并行用时)</b>	1.56499	1.6772	1.8797	3.97293	8.80445
<b>S(加速比)</b>	1.00	0.93	0.83	0.39	0.18
<b>E(效率)</b>	1.00	0.47	0.21	0.05	0.01



#### 5.4 A4 结果

$n$  较小时 ( $n=2, n=4$ ) 时加速比大于 1, 这比 A3 要理想一点 (虽然绝对用时比 A3 更长)。 $n$  增大后用时增加明显, 且加速比、效率都明显下降, 这可能与重复读取量变大有关系, 与算法原理部分的分析相符。

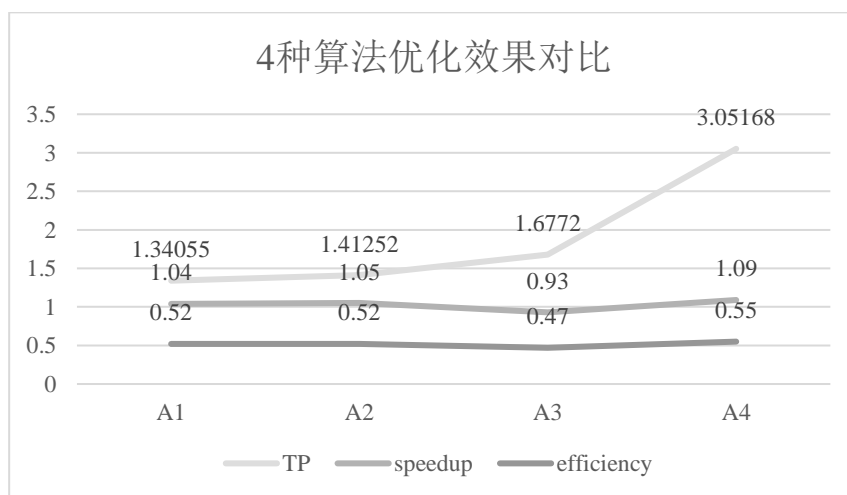
进程数	1	2	4	8	16
$T_p$ (并行用时)	3.34041	3.05168	3.03415	5.73127	12.1261
S(加速比)	1.00	1.09	1.10	0.58	0.28
E(效率)	1.00	0.55	0.28	0.07	0.02



### 5.5 四种算法对比

从上面结果看来，4 种算法的可扩展性都很不理想，所以这里选择  $\text{comm\_sz} = 2$  的结果来对比（这种情况并行优化效果相对好）。

Algorithm	A1	A2	A3	A4
$T_p$ (并行用时)	1.34055	1.41252	1.6772	3.05168
S(加速比)	1.04	1.05	0.93	1.09
E(效率)	0.52	0.52	0.47	0.55



## 6 反思与心得

这次实验中，我对“读取矩阵 **A**”的过程尝试了 4 种不同的并行优化，实验结果说明这种优化的效果都不太明显（加速比仅比 1 大一点），而且它们的可扩展性都不好（效率随着并行进程数增大而下降）。

实验中一个有些困难但很有趣的问题是应该在程序的哪些位置设置 **barrier**。因为在实际运行的时候，我们不知道各个进程工作的顺序，所以我需要在一些阶段性工作完成之后设置 **barrier** 确保各进程都准备好进行下一阶段的工作了。

这次实验中还涉及了将 **MPI** 和 **pthread** 组合使用。启动线程的时候，**create\_thread** 函数可以给线程要运行的函数传递一个参数指针，当线程要运行的函数的参数数量多且类型不同的时候，用一个参数指针来传递它们有些困难，所以我在实际编程的时候更倾向于把它们改成全局变量。

实验 3 中我使用 C 语言编程，这次我改用 C++ 重写了全部代码，编译命令由“**mpicc**”改成“**mpic++**”。