姓名：李其融

学号：21307110403

MPI实现Parallel sorting by regular sampling算法

实验要求

1. 使用Socket或MPI实现Parallel sorting by regular sampling算法
2. 测试程序在1K、5K、10K、100K等不同数据量以及在不同线程数情况下的加速比
3. 可以使用其他语言的以及其他线程库
4. 但是不可以直接调用库函数或者开源项目提供的直接可以完成Parallel sorting by regular sampling算法的接口

实现思路

1. **MIMD机器上PSRS排序算法：**

```
输入：长度为n的无序序列，p台处理器，每台处理器有n/p个元素
输出：长度为n的有序序列
Begin
1．均匀划分：n个元素均匀地划分成p段，每台处理器有n/p个元素
2．局部排序：各处理器利用串行排序算法，排序n/p个数
3．正则采样：每台处理器各从自己地有序段中选取p个样本元素
4．样本排序：用一台处理器将所有p*p个样本元素用串行排序算法排序之
5．选择主元：用一台处理器选取p-1个主元，并将其播送给其余处理器
6．主元划分：各处理器按主元将各自地有序段划分成p段
7．全局交换：各处理器将其辖段按段号交换到相应地处理器中
8．归并排序：处理器使用归并排序将所接收地诸段施行排序
End
```

2. 先在主进程中，采用串行快速排序（参考实验一）对随机数列排序并计时

3. 再按照上面的算法，对相同地随机数列进行排序并计时，算法最后一步结束之后，还需要把各个处理器的段依次回收到主进程，得到最终完整的排好序的数列。

实现过程

生成随机数列，并对串行算法计时

在主进程中随机生成长度为`len`的数列，采用串行的快速排序算法对这个数列的副本进行排序并计时，输出并正确释放内存空间。

```
if (myid == 0)
{
    array = (int *)calloc(len, sizeof(int));
    int *array_serial = NULL;
    array_serial = (int *)calloc(len, sizeof(int));
    srand(time(NULL));
    for (i = 0; i < len; i++)
    {
        array[i] = rand();
        array_serial[i] = array[i];
        // printf("array[%d] = %d\t", i, array[i]);
    }
    double start1, end1;
    start1 = MPI_Wtime();
    quicksort(array_serial, 0, len-1);
    end1 = MPI_Wtime();
    printf("Runtime(serial):   %lf\n", end1 - start1);
    free(array_serial);
}
```

(1) 均匀划分`

`len`个元素均匀地划分成`numprocs`段，每台处理器有len/numprocs个元素，使用`MPI_Scatter()`将数列分成`len_per_proc`个段，所有进程各收到其中一个段

```
int len_per_proc = len / numprocs;
int *a = (int *)calloc(len_per_proc, sizeof(int));
MPI_Scatter(array, len_per_proc, MPI_INT, a, len_per_proc, MPI_INT, 0, MPI_COMM_WORLD);
```

(2) 局部排序

各处理器利用串行排序算法（此处采用串行快速排序），排序len_per_proc个元素。quicksort()是自行实现的快速排序算法，关键代码在实验一的实验报告中有说明，但不是本实验的重点

```
quicksort(a, 0, len_per_proc-1);
```

(3) 正则采样

每台处理器各从自己的有序段中（从第0个元素开始，等间距地）选取numprocs个样本元素

```
int *samples = (int *)calloc(numprocs, sizeof(int));
for (i = 0; i < numprocs; i++)
{
    samples[i] = a[i * numprocs];
}
```

(4) 样本排序

主进程将所有numprocs*numprocs个样本元素用MPI_Cather()收集起来，并用串行算法（此处采用串行快速排序quicksort()）排序之

```
int *samples_all;
if (myid == 0)
{
    samples_all = (int *)calloc((numprocs * numprocs), sizeof(int));
}
MPI_Gather(samples, numprocs, MPI_INT, samples_all, numprocs, MPI_INT, 0, MPI_COMM_WORLD);
if (myid == 0)
{
    quicksort(samples_all, 0, (numprocs*numprocs) - 1);
}
```

(5) 选择主元

主进程从排好序的numprocs*numprocs个样本元素中（从第numproc+1个元素开始选，等间距地）选取numprocs-1个主元，并使用MPI_Bcast()将其播送给其余处理器

```
int *pivots = (int *)calloc((numprocs - 1), sizeof(int));
if (myid == 0)
{
    for (i = 0; i < (numprocs - 1); i++)
    {
        pivots[i] = samples_all[(i + 1) * numprocs];
    }
}
MPI_Bcast(pivots, (numprocs - 1), MPI_INT, 0, MPI_COMM_WORLD);
```

(6) 主元划分

各处理器按主元将各自的有序段划分成numprocs个段，用partition_size记录每一个段地长度，用send_dis记录每一个段的第一个元素的下标

```
int index = 0;
int *partition_size = (int *)calloc(numprocs, sizeof(int)); // 记录每段的长度
int *send_dis = (int *)calloc(numprocs, sizeof(int)); // 记录每段第一个元素的下标
send_dis[0] = 0;
for (i = 0; i < len_per_proc; i++)
```

```
    {
        if (a[i] > pivots[index])
        {
            index += 1;
            send_dis[index] = i;
        }
        if (index == (numprocs ))
        {
            partition_size[index-1] = len_per_proc - i+1;
            send_dis[index] = i;
            break;
        }
        partition_size[index]++;
    }
```

(7) 全局交换

每个处理器先用MPI_Alltoall()将每一段的长度传递到相应的处理器，例如0号处理器收到每个处理器发来的partition_size[0]，1号处理器收到每个处理器发来的partition_size[1]。

因为每个处理器都会各自收到numprocs个新段并组成一个新数列，所以用new_partitions保存新数列，并记录它的长度totalsize，用recv_dis记录新数列的每段的第一个元素的下标，end_dis记录新数列的每段的最后一个元素的下标。

最后每个处理器用MPI_Alltoallv()把自己排好序的数列分段发给其余处理器，同时按照新数列的划分方式从其余处理器接收新的数列。

```
    int *new_partition_size = (int *)calloc(numprocs, sizeof(int));
    MPI_Alltoall(partition_size, 1, MPI_INT, new_partition_size, 1, MPI_INT, MPI_COMM_WORLD);
    int totalsize = 0; // 每个处理器负责归并排序的数组总长度
    for (i = 0; i < numprocs; i++)
        totalsize += new_partition_size[i];
    int *new_partitions = (int *)calloc(totalsize, sizeof(int)); // 每个处理器负责归并排序的数组
    int *recv_dis = (int *)calloc(numprocs, sizeof(int)); // 记录新的每段第一个元素的下标
    int *end_dis = (int *)calloc(numprocs, sizeof(int)); // 记录新的每段最后一个元素的下标
    recv_dis[0] = 0;
    for (i = 1; i < numprocs; i++)
    {
        recv_dis[i] = recv_dis[i - 1] + new_partition_size[i - 1];
        end_dis[i - 1] = recv_dis[i]-1;
    }
    end_dis[numprocs-1] = totalsize-1;
    MPI_Alltoallv(a, partition_size, send_dis, MPI_INT, new_partitions, new_partition_size, recv_dis, MPI_INT,
MPI_COMM_WORLD);
```

(8) 归并排序

处理器使用归并排序将所接收的诸段施行排序

```
    int *sorted_partitions = (int *)calloc(totalsize, sizeof(int));
    for (i = 0; i < totalsize; i++)
    {
        int lowest = INT_MAX;
        int ind = -1;
        for (j = 0; j < numprocs; j++)
        {
            if ((recv_dis[j] <= end_dis[j]) && (new_partitions[recv_dis[j]] < lowest))
            {
                lowest = new_partitions[recv_dis[j]];
                ind = j;
            }
        }
        sorted_partitions[i] = lowest;
        recv_dis[ind] += 1;
    }
```

主进程依次回收排好序的各段

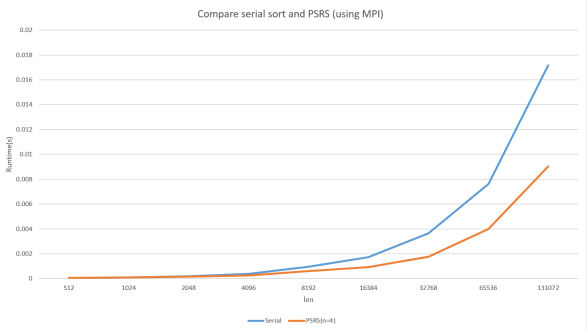
主进程先用`MPI_Gather()`统计每个处理器的数列长度，保存在`recv_count`中，然后再用`MPI_Gatherv()`按顺序从各个处理器中回收相应长度的排好序的数列。

```
int *recv_count; // 主进程中记录每一段的长度
if (myid == 0)
{
    recv_count = (int *)calloc(numprocs, sizeof(int));
}
MPI_Gather(&totalsize, 1, MPI_INT, recv_count, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (myid == 0)
{
    recv_dis[0] = 0;
    for (i = 1; i < numprocs; i++)
        recv_dis[i] = recv_dis[i - 1] + recv_count[i - 1];
}
MPI_Gatherv(sorted_partitions, totalsize, MPI_INT, array, recv_count, recv_dis, MPI_INT, 0, MPI_COMM_WORLD);
```

测试结果与分析

固定处理器数量n=4，改变数列长度`len`：2^{9}、2^{10}、2^{11}、2^{12}、2^{13}、2^{14}、2^{15}、2^{16}、2^{17}，测量串行快速排序和MPI实现的PSRS各自需要的时间，每组实验各运行5次求平均值，最后计算加速比。

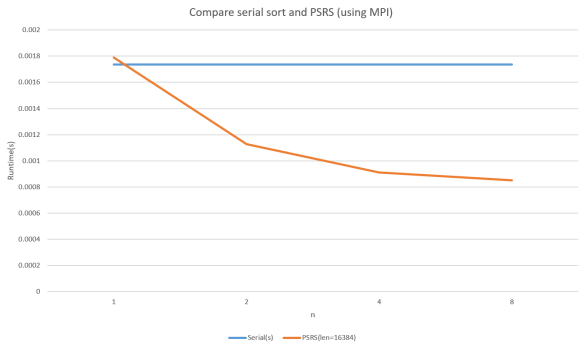
len	512	1024	2048	4096	8192	16384	32768	65536	131072
serial(s)	0.0000404	0.0000956	0.0001852	0.000388	0.0009686	0.0017358	0.0036656	0.0076082	0.017183
parall(s)	0.000075	0.0001066	0.0001698	0.0002642	0.0006096	0.0009116	0.0017584	0.0040146	0.0090358
speedup ratio	0.538666667	0.896810507	1.090694935	1.468584406	1.588910761	1.904124616	2.084622384	1.895132765	1.90165785



可以看出，当数据量较小时，由于各处理器之间的通讯有开销，所以PSRS的效果不如串行，当数据量增大时，PSRS要好于串行。但是当数据量逐渐增大，加速比的增长速度减慢，这是因为MPI实现PSRS的第（7）步全局交换时，如果数据量越大，通讯开销越大。

固定数列长度len=2^{14}，改变处理器数量`n`：1、2、3、4、5、6、7、8，测量串行快速排序和MPI实现的PSRS各自需要的时间，每组实验各运行5次求平均值，最后计算加速比。

n	1	2	4	8
serial(s)	0.0017358	0.0017358	0.0017358	0.0017358
parall(s)	0.001788	0.0011266	0.0009116	0.0008522
speedup ratio	0.970805369	1.540742056	1.904124616	2.036845811



可以看出，处理器数量越多，PSRS需要的时间越少，但是随着处理器数量的增多，加速比的增长速度减慢，这是因为处理器越多通讯开销越大。

综合所有实验结果，MPI实现PSRS比串行快速排序可以快大约2倍，提高了排序的速度。