实验4

实验环境

macOS 10.14，mpirun (Open MPI) 4.0.3，g++-9 (Homebrew GCC 9.3.0_1) 9.3.0

由于电脑线程不够（仅有2核4线程），在使用mpi时使用了`oversubscribe`，所以有部分无法运行。

算法设计与分析

该算法完全依照说明。

核心代码

```
void PSRS(int* data)
{
    double startTime = MPI_Wtime();
    int size, size2, size1, rank, *partitionCount, *rPartitionCount;
    int subArraySize, startIdx, endIdx, * sample, * rPartitions;
    int localN = N / np;
    int gap = localN / np;

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    size2 = size * size;
    sample = (int*)malloc(size2 * sizeof(int));
    partitionCount = (int*)malloc(size * sizeof(int));
    rPartitionCount = (int*)malloc(size * sizeof(int));

    for (int i = 0; i < size; i++)
    {
        partitionCount[i] = 0;
    }

    startIdx = rank * N / size;
    if (size == (rank + 1))
    {
        endIdx = N;
    }
    else
    {
        endIdx = (rank + 1) * N / size;
    }
```

```
    subArraySize = endIdx - startIdx;

    MPI_Barrier(MPI_COMM_WORLD);

    // 对子数组进行局部排序
    qsort(data + startIdx, subArraySize, sizeof(data[0]), cmp);

    // 正则采样
    for (int i = 0; i < size; i++)
    {
        int idx = rank * localN + i * gap;
        sample[rank * np + i] = *(data + idx);
    }

    size1 = size - 1;
    int* pivot = (int*)malloc((size1) * sizeof(sample[0])); //主元
    int index = 0;

    MPI_Barrier(MPI_COMM_WORLD);

    if (rank == 0)
    {
        qsort(sample, size, sizeof(sample[0]), cmp); //对正则采样的样本进行排序

        // 采样排序后进行主元的选择
        for (int i = 0; i < (size - 1); i++)
        {
            pivot[i] = sample[(((i + 1) * size) + (size / 2)) - 1];
        }
    }

    //发送广播
    MPI_Bcast(pivot, size - 1, MPI_INT, 0, MPI_COMM_WORLD);

    // 主元划分
    for (int i = 0; i < subArraySize; i++)
    {
        if (data[startIdx + i] > pivot[index])
        {
            index += 1;
        }
        if (index == size)
        {
            partitionCount[size - 1] = subArraySize - i + 1;
            break;
        }
```

```
        partitionCount[index]++; //划分大小自增
    }
    free(pivot);

    int count = 0;
    int* sdispls = (int*)malloc(size * sizeof(int));
    int* rdispls = (int*)malloc(size * sizeof(int));

    // 全局到全局的发送
    MPI_Alltoall(partitionCount, 1, MPI_INT, rPartitionCount, 1, MPI_INT,
MPI_COMM_WORLD);

    // 计算划分的总大小，并给新划分分配空间
    for (int i = 0; i < size; i++)
    {
        count += rPartitionCount[i];
    }
    rPartitions = (int*)malloc(count * sizeof(int));

    sdispls[0] = 0;
    rdispls[0] = 0;
    for (int i = 1; i < size; i++)
    {
        sdispls[i] = partitionCount[i - 1] + sdispls[i - 1];
        rdispls[i] = rPartitionCount[i - 1] + rdispls[i - 1];
    }

    //发送数据，实现n次点对点通信
    MPI_Alltoallv(&(data[startIdx]), partitionCount, sdispls, MPI_INT,
rPartitions, rPartitionCount, rdispls, MPI_INT, MPI_COMM_WORLD);

    free(sdispls);
    free(rdispls);

    SortSub(rPartitions, rPartitionCount, size, rank, data);

    double endTime = MPI_Wtime();

    if (rank == 0)
    {
        cout << "Sorted:"<<endl;
        for (int i = 0; i < N; i++)
        {
            cout << data[i] << " ";
        }
        cout << endl;
```

```
        cout << "np:" << np << endl;
        printf("%lf\n", endTime - startTime);
    }

    if (size > 1)
    {
        free(rPartitions);
    }
    free(partitionCount);
    free(rPartitionCount);
    free(sample);
    free(data);

    MPI_Finalize();
}
```

实验结果

```
Original:
16807 282475249 622650073 984943658 144108930 470211272 101027544 45785087
8 458777923 7237709 823564440 115438165 784484492 74243042 114807987 13752
2503 441282327 16531729 823378840 143542612 896544303 474833169 264817709
998097157 817129560 131570933 197493099 404280278 893351816 505795335 9548
99097 636807826 563613512 101929267 580723810 704877633 358580979 62437914
9 128236579 784558821 530511967 110010672 551901393 617819336 399125485 15
6091745 356425228 899894091 585640194 937186357 646035001 25921153 5106167
08 590357944 771515668 357571490 44788124 927702196 952509530 130060903 94
2727722 83454666 108728549 685118024
Sorted:
16807 7237709 16531729 25921153 44788124 74243042 83454666 101027544 10192
9267 108728549 110010672 114807987 115438165 128236579 130060903 131570933
 137522503 143542612 144108930 156091745 197493099 264817709 282475249 356
425228 357571490 358580979 399125485 404280278 441282327 457850878 4587779
23 470211272 474833169 505795335 510616708 530511967 551901393 563613512 5
80723810 585640194 590357944 617819336 622650073 624379149 636807826 64603
5001 685118024 704877633 771515668 784484492 784558821 817129560 823378840
 823564440 893351816 896544303 899894091 927702196 937186357 942727722 952
```

64个数排序结果如上。

我又对两个较大规模的数据进行了测试。

规模/进程数	1	2	4	8
1,000,000	0.196362	0.122315/1.60538	0.104532/1.87849	0.154368/1.27204
10,000,000	2.208989	1.424954/1.55022	1.270483/1.7387	1.621450/1.36235

运行时间(sec)/加速比

分析与总结

在我的环境中，四线程获得了最好的结果。但是加速比无法突破2，可能是对$O(n^2)$个节点进行通信操作开销太大。