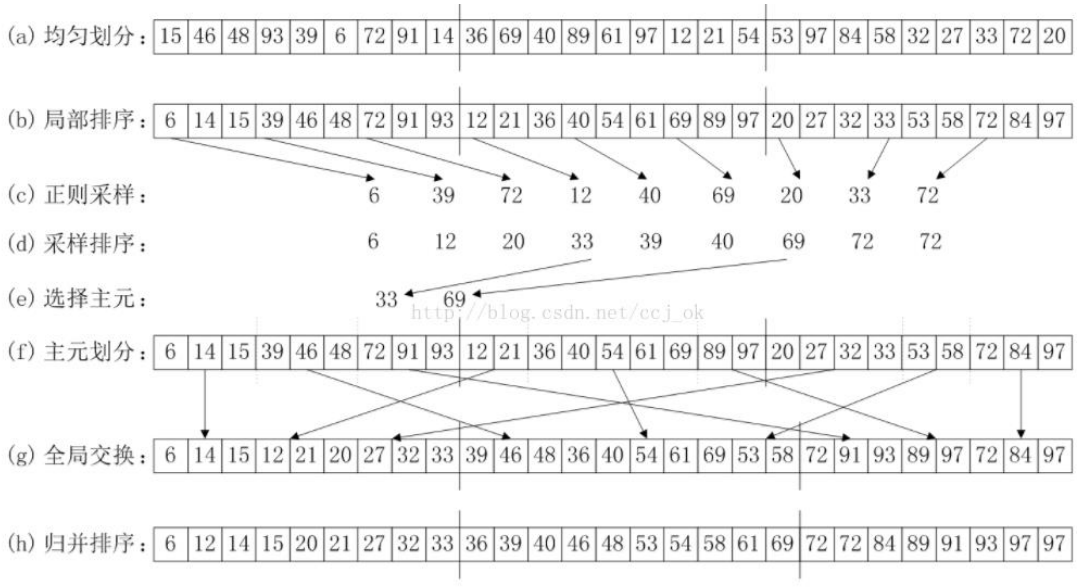
实验 4 实验报告

实验题目：利用 MPI 实现并行排序算法

实验环境: 实验所使用电脑操作系统为 windows10 操作系统， IDE 为 Visual Studio2017，OpenMP 实验环境由 VS2017 提供， MPI 使用 MS-MPI, 代码编写和测试均由 VS2017 实现。

算法设计与分析:
首先我们要明确并行排序算法的过程:
(1)均匀划分：将 n 个元素 A[1..n]均匀划分成 p 段，每个 pi 处理 A[(i-1)n/p+1..in/p]
(2)局部排序：pi 调用串行排序算法对 A[(i-1)n/p+1..in/p]排序
(3)选取样本：pi 从其有序子序列 A[(i-1)n/p+1..in/p]中选取 p 个样本元素
(4)样本排序：用一台处理器对 p2 个样本元素进行串行排序
(5)选择主元：用一台处理器从排好序的样本序列中选取 p-1 个主元，并播送给其他 pi
(6)主元划分： pi 按主元将有序段 A[(i-1)n/p+1..in/p]划分成 p 段
(7)全局交换：各处理器将其有序段按段号交换到对应的处理器中
(8)归并排序：各处理器对接收到的元素进行归并排序



本次算法实现严格按照并行排序算法执行步骤编写。使用具体的 MPI 函数接口实现各个进程间通信和同步。

核心代码:

```
pivots = (int *)malloc(p * sizeof(int));
partitionSizes = (int *)malloc(p * sizeof(int));
newPartitionSizes = (int *)malloc(p * sizeof(int));
for (k = 0; k < p; k++) {
    partitionSizes[k] = 0;
}

// 获取起始位置和子数组大小
startIndex = myId * N / p;
if (p == (myId + 1)) {
    endIndex = N;
}
else {
    endIndex = (myId + 1) * N / p;
}
subArraySize = endIndex - startIndex;

MPI_Barrier(MPI_COMM_WORLD);

qsort(array + startIndex, subArraySize, sizeof(array[0]), cmp);

for (i = 0; i < p; i++) {
    pivots[i] = array[startIndex + (i * (N / (p * p)))];
}
```

如上图，首先对整个数组进行分组，获取各个进程所分配到的数组数据数。随后先调用库函数对分配到的数据进行排序，然后进行采样获取 p 个采样数据。

```
if (p > 1) {
    collectedPivots = (int *)malloc(p * p * sizeof(pivots[0]));
    phase2Pivots = (int *)malloc((p - 1) * sizeof(pivots[0]));

    MPI_Gather(pivots, p, MPI_INT, collectedPivots, p, MPI_INT, 0, MPI_COMM_WORLD);
    if (myId == 0) {

        qsort(collectedPivots, p * p, sizeof(pivots[0]), cmp);


        for (i = 0; i < (p - 1); i++) {
            phase2Pivots[i] = collectedPivots[(((i + 1) * p) + (p / 2)) - 1];
        }
    }
    //发广播
    MPI_Bcast(phase2Pivots, p - 1, MPI_INT, 0, MPI_COMM_WORLD);

    for (i = 0; i < subArraySize; i++) {
        if (array[startIndex + i] > phase2Pivots[index]) {
            index += 1;

        }
        if (index == p) {
            partitionSizes[p - 1] = subArraySize - i + 1;
            break;
        }
        partitionSizes[index]++;
    }
```

随后在并行的情况下，所有进程把采样数据发送给根进程，根进程负责把这 p^2 个采样点进行排序，随后从其中选取 p-1 个数据作为分割点广播出去。

每个进程获取到分割点后，按照分割点把自己负责的数据进行分割，并记录每个分割的大小。

```
sendDisp = (int *)malloc(p * sizeof(int));
recvDisp = (int *)malloc(p * sizeof(int));

// 全局发送到全局
MPI_Alltoall(partitionSizes, 1, MPI_INT, newPartitionSizes, 1, MPI_INT, MPI_COMM_WORLD);

for (i = 0; i < p; i++) {
    totalSize += newPartitionSizes[i];
}
newPartitions = (int *)malloc(totalSize * sizeof(int));


sendDisp[0] = 0;
recvDisp[0] = 0;
for (i = 1; i < p; i++) {
    sendDisp[i] = partitionSizes[i - 1] + sendDisp[i - 1];
    recvDisp[i] = newPartitionSizes[i - 1] + recvDisp[i - 1];
}

//发送数据，实现n次点对点通信
MPI_Alltoallv(&(array[startIndex]), partitionSizes, sendDisp, MPI_INT, newPartitions, newPartitionSizes, recvDisp, MPI_INT, MPI_COMM_WORLD);
```

首先调用 MPI_Alltoall()函数使得每个进程都知道自己即将接受的 p-1 段数据（不算自己的）的规模。求和计算总的数据规模。在各个进程都明确自己将要发送和接收的数据和数据的规模后使用 MPI_Alltoallv()函数，使得进程 i 向进程 j 发送自己分割得到的第 j 段数据，并接收其他 p-1 个进程发给自己的数据段。

```
// 归并排序
for (i = 0; i < totalListSize; i++) {
    int lowest = INT_MAX;
    int ind = -1;
    for (j = 0; j < p; j++) {
        if ((indexes[j] < partitionEnds[j]) && (newPartitions[indexes[j]] < lowest)) {
            lowest = newPartitions[indexes[j]];
            ind = j;
        }
    }
    sortedSubList[i] = lowest;
    indexes[ind] += 1;
}
```

随后每个处理器对自己的 p 段新的数据进行归并排序。

```
MPI_Gather(&totalListSize, 1, MPI_INT, subListSizes, 1, MPI_INT, 0, MPI_COMM_WORLD);


if (myId == 0) {
    recvDisp[0] = 0;
    for (i = 1; i < p; i++) {
        recvDisp[i] = subListSizes[i - 1] + recvDisp[i - 1];
    }
}

MPI_Gatherv(sortedSubList, totalListSize, MPI_INT, array, subListSizes, recvDisp, MPI_INT, 0, MPI_COMM_WORLD);
```

最后各个进程把自己的已排序完成的数据规模发送给根进程，然后根进程按照接收到的数据规模信息进行数据的接收和存放。完成排序。

实验结果：
使用 freopen()函数把命令行输出重定向到文件：

```
Process 0 is on LAPTOP-5IHOIRQM
365 1031 1216 1435 1580 1598 1702 1825 3768 4195 4482 5347 5415 5629 5637 6401 6470 7121 7628 7851 9540 10254 10524 10633 11254 11409 11529 11605 11875 12136
 12245 13026 13079 15274 15489 15646 15810 16513 16704 17112 17251 17391 17635 17688 18121 18834 18993 19574 19867 20443 22229 23209 23830 24504 24698 25262
 25857 25981 28980 29816 29982 31753 32505 32578

time:0.087000
```

运行时间

规模/并行度	1	2	3	4
10000000	4.725s	3.092s	2.252s	2.798s

加速比：

规模/并行度	1	2	3	4
10000000	1	1.528	2.098	1.689

实验总结与分析：

并行实验的难点就是进程间的相互通信，这次的通信中使用到的 MPI_Gather(), MPI_Alltoall(), MPI_Alltoallv()都需要深刻理解函数含义才能正确使用。

分析这次实验的实验和测试结果容易知道，并行编程的优越性常常在数据规模较大时才能展现出性能方面的优越性。且随着并行度的增加，进程间通信成本也在逐渐攀升，因此我们必须具体情况具体分析，在需要把并行编程应用到工作和生活中时，根据实际情况选择串行编程和并行编程，根据具体情况选择并行编程下的并行度，使得效率达到最高。例如在这次实验排序一千万数据时，在并行度从 1 开始增加时，运行时间在逐步减小，但当并行度从 3 上升到 4 时，通信成本增加使得排序的性能低于并行度为 3 时候的情况。