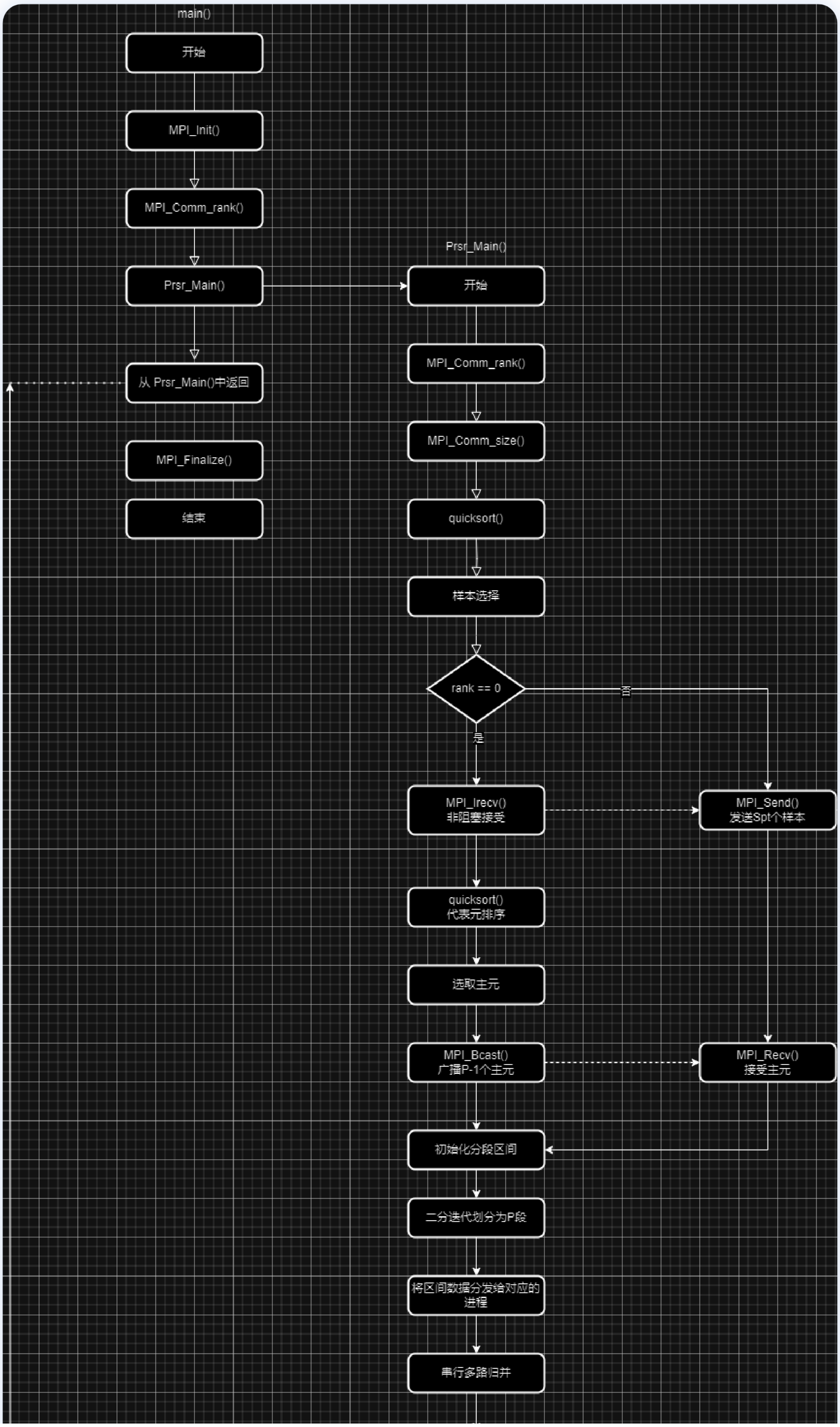
并行程序第二次作业

21373365 范吴运维

JobID: 8448037

流程图


main()
开始
MPI_Init()
MPI_Comm_rank()
Prsr_Main()
从 Prsr_Main()中返回
MPI_Finalize()
结束
Prsr_Main()
开始
MPI_Comm_rank()
MPI_Comm_size()
quicksort()
样本选择
rank == 0
否
是
MPI_Irecv()
非阻塞接受
MPI_Send()
发送Spt个样本
quicksort()
代表元排序
选取主元
MPI_Bcast()
广播P-1个主元
MPI_Recv()
接受主元
初始化分段区间
二分迭代划分为P段
将区间数据分发给对应的进程
串行多路归并

代码注释

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define INIT_TYPE 10
#define ALLTOONE_TYPE 100
#define ONETOALL_TYPE 200
#define MULTI_TYPE 300
#define RESULT_TYPE 400
#define RESULT_LEN 500
#define MULTI_LEN 600

#define LOCAL_ENV
#define DEBUG_MODE

#define seg_st(x) (2 * (x))
#define seg_ed(x) (2 * (x) + 1)

int Spt;        // Sample per thread 样本数量
long DataSize; // 数据集大小
int *num, *arr1;
int mylength; // 单个进程的长度
int *index;
int *samples;

int main(int argc, char *argv[])
{
    long BaseNum = 1;
    int MultNum;
    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    MultNum = 6 * size * size;
    DataSize = BaseNum * MultNum;

    if (rank == 0)
        printf("The DataSize is : %lu\n", DataSize);
    Psrs_Main();

    if (rank == 0)
```

```
        printf("\n");

    MPI_Finalize();
}

void Psrs_Main()
{
    int i, j;
    int rank, size;
    int n, lft, mid, rht, cur, k, l;
    FILE *fp;
    int ready;
    MPI_Status status[32 * 32 * 2];
    MPI_Request request[32 * 32 * 2];

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

    // Samples per thread, 每个线程的样本数
    Spt = size - 1;

    /*初始化参数*/
    num = (int *)malloc(2 * DataSize * sizeof(int));
    if (num == 0)
        merror("malloc memory for arr error!");
    arr1 = &num[DataSize];

    if (size > 1)
    {
        // size > 1 当启用并行的情况
        // 当进程数超过1时，申请 size *  Spt 个int 的内存， 用于存储代样本
        samples = (int *)malloc(sizeof(int) * size * Spt);
        if (samples == 0)
            merror("malloc memory for temp1 error!");
        // index : 2* size 个 int
        index = (int *)malloc(sizeof(int) * 2 * size);
        if (index == 0)
            merror("malloc memory for index error!");
    }

#ifdef LOCAL_ENV
    static char fileName[10];
    sprintf(fileName, "./%d.out", rank);
    freopen(fileName, "w", stdout);
#endif
```

```
    MPI_Barrier(MPI_COMM_WORLD);
    // 均摊的数据量，不一定能整除
    mylength = DataSize / size;
    srand(rank);

    printf("This is node %d , mylength is %d\n", rank, mylength);
    printf("On node %d the input data is:\n", rank);

    for (i = 0; i < mylength; i++)
    {
        num[i] = (int)rand();
        printf("%d\n", num[i]);
    }
    printf("\n");

    /*每个处理器将自己的n/P个数据用串行快速排序(Quicksort)，得到一个排好序的序列，对应于算
法13.5步骤 (1) */
    MPI_Barrier(MPI_COMM_WORLD);
    quicksort(num, 0, mylength - 1);
    MPI_Barrier(MPI_COMM_WORLD);

#ifdef DEBUG_MODE
    printf("After Sort:\n");
    int t;
    for (t = 0; t < mylength; t++)
    {
        printf("%d\n", num[t]);
    }
    printf("\n\n");
#endif

    /*每个处理器从排好序的序列中选取第w, 2w, 3w, …, Spt*w个，共Spt个数据作为代表元素，其中
w=mylength/(Spt + 1) 为间隔*/
    if (size > 1)
    {
        MPI_Barrier(MPI_COMM_WORLD);
        // n 即 w，注意不能为0
        n = (int)(mylength / (Spt + 1));
        if (n <= 0)
        {
            merror("width <= zero\n");
        }
        for (i = 0; i < Spt; i++)
            samples[i] = num[(i + 1) * n - 1];

        // 选择主元
        MPI_Barrier(MPI_COMM_WORLD);

        if (rank == 0)
        {
            /*每个处理器将选好的代表元素送到处理器P0中，对应于算法13.5步骤（3） */
            j = 0;
            for (i = 1; i < size; i++)
                // 非阻塞地接收消息
                MPI_Irecv(&samples[i * Spt], sizeof(int) * Spt, MPI_CHAR, i,
ALLTOONE_TYPE + i, MPI_COMM_WORLD, &request[j++]);
            MPI_Waitall(size - 1, request, status);

            /* 处理器P0将上一步送来的P段有序的数据序列做P路归并，再选择排序后的第P-1,
2(size-1), …, (size-1)(size-1)个共P-1个主元，，对应于算法13.5步骤（3）*/
            MPI_Barrier(MPI_COMM_WORLD);
            quicksort(samples, 0, size * Spt - 1);
            MPI_Barrier(MPI_COMM_WORLD);
            // 选择(P-1)个主元
            for (i = 0; i < Spt; i++)
                samples[i] = samples[(i + 1) * Spt - 1];
            /*处理器P0将这P-1个主元播送到所有处理器中，对应于算法13.5步骤（4）*/
            MPI_Bcast(samples, sizeof(int) * Spt, MPI_CHAR, 0, MPI_COMM_WORLD);
            MPI_Barrier(MPI_COMM_WORLD);
        }
        else
        {
            MPI_Send(samples, sizeof(int) * Spt, MPI_CHAR, 0, ALLTOONE_TYPE +
rank, MPI_COMM_WORLD);
            MPI_Barrier(MPI_COMM_WORLD);
            MPI_Barrier(MPI_COMM_WORLD);
            MPI_Bcast(samples, sizeof(int) * Spt, MPI_CHAR, 0, MPI_COMM_WORLD);
            MPI_Barrier(MPI_COMM_WORLD);
        }

        /*每个处理器根据上步送来的P-1个主元把自己的mylength个数据分成P段，记为处理器Pi的第
j+1段，其中i=0,…,size-1，j=0,…,size-1，对应于算法13.5步骤（5）*/
        n = mylength;
        // 初始化 首个区间， 区间取值 [0,Spt]
        // index 维护左闭右开的端点
        index[seg_st(0)] = 0;
        i = 0;
        // 枚举主元 找到第一个满足 num[0] < sample[i] 的区间
        while ((num[0] >= samples[i]) && (i < Spt))
        {
            // 将对应的空间
            index[seg_ed(i)] = 0;
```

```
    index[seg_st(i + 1)] = 0;
    i++;
}

// 停止时若有 i == Spt, 则所有数都落在最右区间
if (i == Spt)
    index[seg_ed(Spt)] = n;

lft = 0;
// 二分查找
while (i < Spt)
{
    cur = samples[i];
    rht = n;
    mid = (int)((lft + rht) / 2);
    while ((num[mid] != cur) && (lft < rht))
    {
        if (num[mid] > cur)
        {
            rht = mid - 1;
            mid = (int)((lft + rht) / 2);
        }
        else
        {
            lft = mid + 1;
            mid = (int)((lft + rht) / 2);
        }
    }

    while ((num[mid] <= cur) && (mid < n))
        mid++;

    // num[mid] 维护第一个大于当前主元的值
    // 同时作为上一个区间右端点和当前区间左端点

    if (mid == n)
    {
        index[seg_ed(i)] = n;
        for (k = i + 1; k < size; k++)
        {
            index[seg_st(k)] = n;
            index[seg_ed(k)] = n;
        }
        i = Spt;
    }
    else
```

```
            {
                index[seg_ed(i)] = mid;
                i++;
                index[seg_st(i)] = mid;
            }
            lft = mid;
            mid = (int)((lft + rht) / 2);
        }

        // 处理最后一个主元
        if (i == Spt)
            index[seg_ed(Spt)] = n;

        MPI_Barrier(MPI_COMM_WORLD);

        /*每个处理器送它的第i+1段给处理器Pi，从而使得第i个处理器含有所有处理器的第i段数据
(i=0,…,size-1)，，对应于算法13.5步骤（6）*/

        j = 0;
        for (i = 0; i < size; i++)
        {
            if (i == rank)
            {
                samples[i] = index[seg_ed(i)] - index[seg_st(i)];
                for (n = 0; n < size; n++)
                    if (n != rank)
                    {
                        // 向其他进程发送
                        k = index[seg_ed(n)] - index[seg_st(n)];
                        MPI_Send(&k, sizeof(int), MPI_CHAR, n, MULTI_LEN +
rank, MPI_COMM_WORLD);
                    }
            }
            else
            {
                MPI_Recv(&samples[i], sizeof(int), MPI_CHAR, i, MULTI_LEN + i,
MPI_COMM_WORLD, &status[j++]);
            }
        }

        MPI_Barrier(MPI_COMM_WORLD);

        j = 0;
        k = 0;
        l = 0;
        for (i = 0; i < size; i++)
        {
            MPI_Barrier(MPI_COMM_WORLD);

            if (i == rank)
            {
                // 将自己的数据拷贝到arr1 中
                for (n = index[seg_st(i)]; n < index[seg_ed(i)]; n++)
                    arr1[k++] = num[n];
            }

            MPI_Barrier(MPI_COMM_WORLD);

            if (i == rank)
            {

                for (n = 0; n < size; n++)
                    if (n != rank)
                    {
                        // 将该区间数据发到对应的进程
                        MPI_Send(&num[index[seg_st(n)]], sizeof(int) *
(index[seg_ed(n)] - index[seg_st(n)]), MPI_CHAR, n, MULTI_TYPE + rank,
MPI_COMM_WORLD);
                    }
            }
            else
            {
                l = samples[i];
                MPI_Recv(&arr1[k], l * sizeof(int), MPI_CHAR, i, MULTI_TYPE +
i, MPI_COMM_WORLD, &status[j++]);
                k = k + l;
            }

            MPI_Barrier(MPI_COMM_WORLD);
        }
        mylength = k;
        MPI_Barrier(MPI_COMM_WORLD);

        /*每个处理器再通过P路归并排序将上一步的到的数据排序；从而这n个数据便是有序的，，对应
于算法13.5步骤（7） */
        k = 0;
        multimerge(arr1, samples, num, &k, size);
        MPI_Barrier(MPI_COMM_WORLD);
    }

    printf("On node %d, length: %d the sorted data is : \n", mylength, rank);
    for (i = 0; i < mylength; i++)
        printf("%d\n", num[i]);
    printf("\n");
}

/*输出错误信息*/
merror(char *ch)
{
    printf("%s\n", ch);
    exit(1);
}

/*串行快速排序算法*/
quicksort(int *datas, int bb, int ee)
{
    int tt, i, j;
    tt = datas[bb];
    i = bb;
    j = ee;

    if (i < j)
    {
        while (i < j)
        {
            while ((i < j) && (tt <= datas[j]))
                j--;
            if (i < j)
            {
                datas[i] = datas[j];
                i++;
                while ((i < j) && (tt > datas[i]))
                    i++;
                if (i < j)
                {
                    datas[j] = datas[i];
                    j--;
                    if (i == j)
                        datas[i] = tt;
                }
                else
                    datas[j] = tt;
            }
            else
                datas[i] = tt;
        }
        quicksort(datas, bb, i - 1);
        quicksort(datas, i + 1, ee);
    }
}

/* 串行多路归并算法
 * data: 待排序的有序数组:
 * len: 有序区间的长度 len[i] ≥ 0
 * ans : 全局有序数组
 * size: 段数
 */

void multimerge(int *data, int *len, int *ans, int *iter, int size)
{
    int i, j, n;

    j = 0;
    for (i = 0; i < size; i++)
        if (len[i] > 0)
        {
            len[j++] = len[i];
            if (j < i + 1)
                len[i] = 0;
        }

    if (j > 1)
    {
        n = 0;
        for (i = 0; i + 1 < j; i = i + 2)
        {
            merge(&(data[n]), len[i], len[i + 1], &(ans[n]));
            len[i] += len[i + 1];
            len[i + 1] = 0;
            n += len[i];
        }
        // 当不为偶数时， 余最后一个
        if (j % 2 == 1)
            for (i = 0; i < len[j - 1]; i++)
            {
                int tmp = data[n];
                ans[n++] = tmp;
            }
        (*iter)++;
        // 递归调用
        memcpy(data, ans, sizeof(int) * n);
        multimerge(data, len, ans, iter, size);
    }
}

/* 归并排序
 * data: 待合并的数组
 * s1: 数组1长度
 * s2: 数组2长度
 * ans: 结果存储
 */
void merge(int *data, int s1, int s2, int *res)
{
    int i, id1, id2;

    id1 = 0;
    id2 = s1;
    for (i = 0; i < s1 + s2; i++)
    {
        if (id1 == s1)
            res[i] = data[id2++];
        else if (id2 == s2 + s1)
            res[i] = data[id1++];
        else if (data[id1] > data[id2])
            res[i] = data[id2++];
        else
            res[i] = data[id1++];
    }
}
```

执行脚本

```bash
#!/bin/bash
#SBATCH -J waysome_hw2   #作业名
#SBATCH -p cpu-quota
#SBATCH -N 4            #4 节点
#SBATCH -n 4           #28 核
#SBATCH -o waysome_hw2.out # 将标准输出结果
#SBATCH -e waysome_hw2.err # 将错误输出结果

srun hostname | sort > machinefile.${SLURM_JOB_ID}
NP=`cat machinefile.${SLURM_JOB_ID} | wc -l`
module load intel/19.0.5.281
export I_MPI_HYDRA_TOPOLIB=ipl
mpirun -genv I_MPI_FABRICS shm:dapl -np ${NP} -f ./machinefile.${SLURM_JOB_ID}
./tes
```

实验结果

0.out

```
This is node 0 , mylength is 24
On node 0 the input data is:
1804289383
846930886
1681692777
1714636915
1957747793
424238335
719885386
1649760492
596516649
1189641421
1025202362
1350490027
783368690
1102520059
2044897763
1967513926
1365180540
1540383426
304089172
1303455736
```

```
35005211
521595368
294702567
1726956429

After Sort:
35005211
294702567
304089172
424238335
521595368
596516649
719885386
783368690
846930886
1025202362
1102520059
1189641421
1303455736
1350490027
1365180540
1540383426
1649760492
1681692777
1714636915
1726956429
1804289383
1957747793
1967513926
2044897763


On node 29, length: 0 the sorted data is :
8614858
21468264
35005211
35005211
87517201
126313438
142559277
149585093
190686788
260874575
294702567
294702567
304089172
```

```
304089172
310914940
374612515
378651393
396476315
424238335
424238335
483147985
495649264
521595368
521595368
552076975
582691149
591232730
596516649
596516649
```

1.out

```
This is node 1 , mylength is 24
On node 1 the input data is:
1804289383
846930886
1681692777
1714636915
1957747793
424238335
719885386
1649760492
596516649
1189641421
1025202362
1350490027
783368690
1102520059
2044897763
1967513926
1365180540
1540383426
304089172
1303455736
35005211
```

```
521595368
294702567
1726956429

After Sort:
35005211
294702567
304089172
424238335
521595368
596516649
719885386
783368690
846930886
1025202362
1102520059
1189641421
1303455736
1350490027
1365180540
1540383426
1649760492
1681692777
1714636915
1726956429
1804289383
1957747793
1967513926
2044897763


On node 14, length: 1 the sorted data is :
612121425
635050179
693014654
719885386
719885386
747983061
783368690
783368690
787097142
820715049
844158168
846930886
846930886
906156498
```

2.out

```
This is node 2 , mylength is 24
On node 2 the input data is:
1505335290
1738766719
190686788
260874575
747983061
906156498
1502820864
142559277
1261608745
1380759627
2127304342
635050179
582691149
149585093
2039335037
820715049
693014654
2122498773
1809302367
591232730
1281246002
1194903572
1820868569
396476315

After Sort:
142559277
149585093
190686788
260874575
396476315
582691149
591232730
635050179
693014654
747983061
820715049
906156498
```

```
1194903572
1261608745
1281246002
1380759627
1502820864
1505335290
1738766719
1809302367
1820868569
2039335037
2122498773
2127304342


On node 12, length: 2 the sorted data is :
953350440
953369895
989089924
1025202362
1025202362
1102520059
1102520059
1189641421
1189641421
1194903572
1205554746
1207815258
```

3.out

```
This is node 3 , mylength is 24
On node 3 the input data is:
1205554746
483147985
844158168
953350440
612121425
310914940
1210224072
1856883376
1922860801
495649264
8614858
```

```
989089924
378651393
1344681739
2029100602
1816952841
21468264
552076975
87517201
953369895
374612515
787097142
126313438
1207815258

After Sort:
8614858
21468264
87517201
126313438
310914940
374612515
378651393
483147985
495649264
552076975
612121425
787097142
844158168
953350440
953369895
989089924
1205554746
1207815258
1210224072
1344681739
1816952841
1856883376
1922860801
2029100602


On node 41, length: 3 the sorted data is :
1210224072
1261608745
1281246002
1303455736
```

```
1303455736
1344681739
1350490027
1350490027
1365180540
1365180540
1380759627
1502820864
1505335290
1540383426
1540383426
1649760492
1649760492
1681692777
1681692777
1714636915
1714636915
1726956429
1726956429
1738766719
1804289383
1804289383
1809302367
1816952841
1820868569
1856883376
1922860801
1957747793
1957747793
1967513926
1967513926
2029100602
2039335037
2044897763
2044897763
2122498773
2127304342
```