# Parallel quicksort

Name: Shiyao Yu

ID: fd9286

## Abstract

This is the project report of the Parallel Course of Wayne State University. The goal is to get a higher acknowledge of parallel system with MPI and to practice the ability of coding in parallel system.

## Introduction

The quicksort formulation for a shared-address-space system works as follows. Let *A* be an array of *n* elements that need to be sorted and *p* be the number of processes. Each process is assigned a consecutive block of *n*/*p* elements, and the labels of the processes define the global order of the sorted sequence. Let *Ai* be the block of elements assigned to process *Pi* .

The algorithm starts by selecting a pivot element, which is broadcast to all processes. Each process *Pi*, upon receiving the pivot, rearranges its assigned block of elements into two sub- blocks, one with elements smaller than the pivot *Si* and one with elements larger than the pivot *Li*. This *local* rearrangement is done in place using the *collapsing the loops* approach of quicksort. The next step of the algorithm is to rearrange the elements of the original array *A* so

that all the elements that are smaller than the pivot (i.e., ) are stored at the beginning

of the array, and all the elements that are larger than the pivot (i.e., ) are stored at the end of the array.

Once this *global* rearrangement is done, then the algorithm proceeds to partition the processes into two groups, and assign to the first group the task of sorting the smaller elements *S*, and to the second group the task of sorting the larger elements *L* . Each of these steps is performed by recursively calling the parallel quicksort algorithm. Note that by simultaneously partitioning both the processes and the original array each group of processes can proceed independently. The recursion ends when a particular sub-block of elements is assigned to only a single process, in which case the process sorts the elements using a serial quicksort algorithm.

The partitioning of processes into two groups is done according to the relative sizes of the *S* and

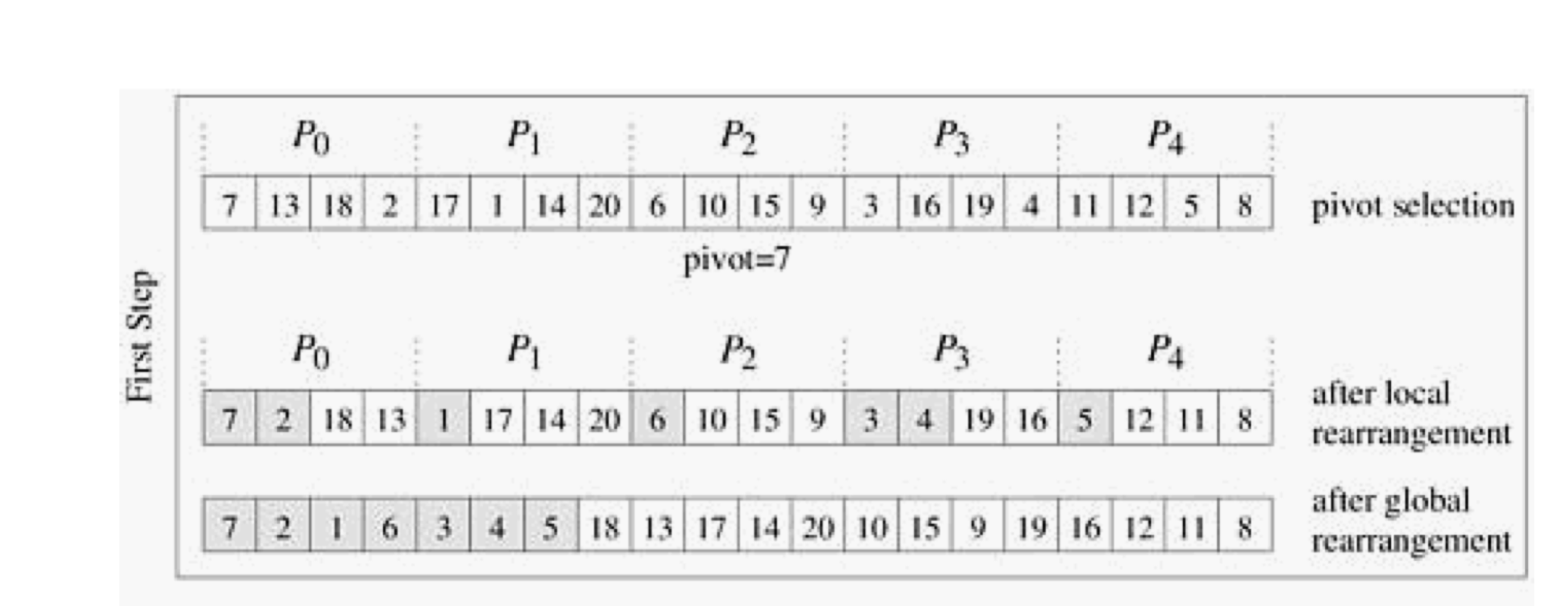
*L* blocks. In particular, the first processes are assigned to sort the smaller elements *S*, and the rest of the processes are assigned to sort the larger elements *L*. Note that

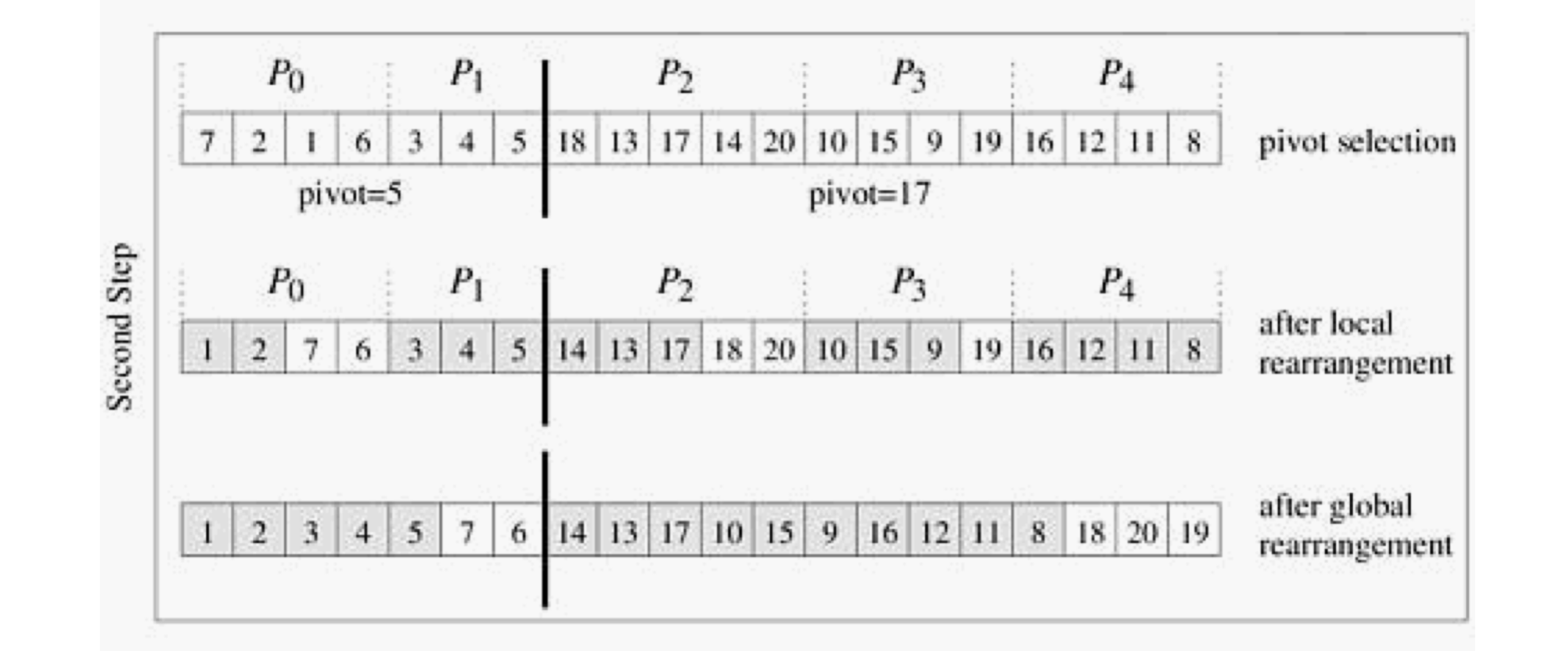


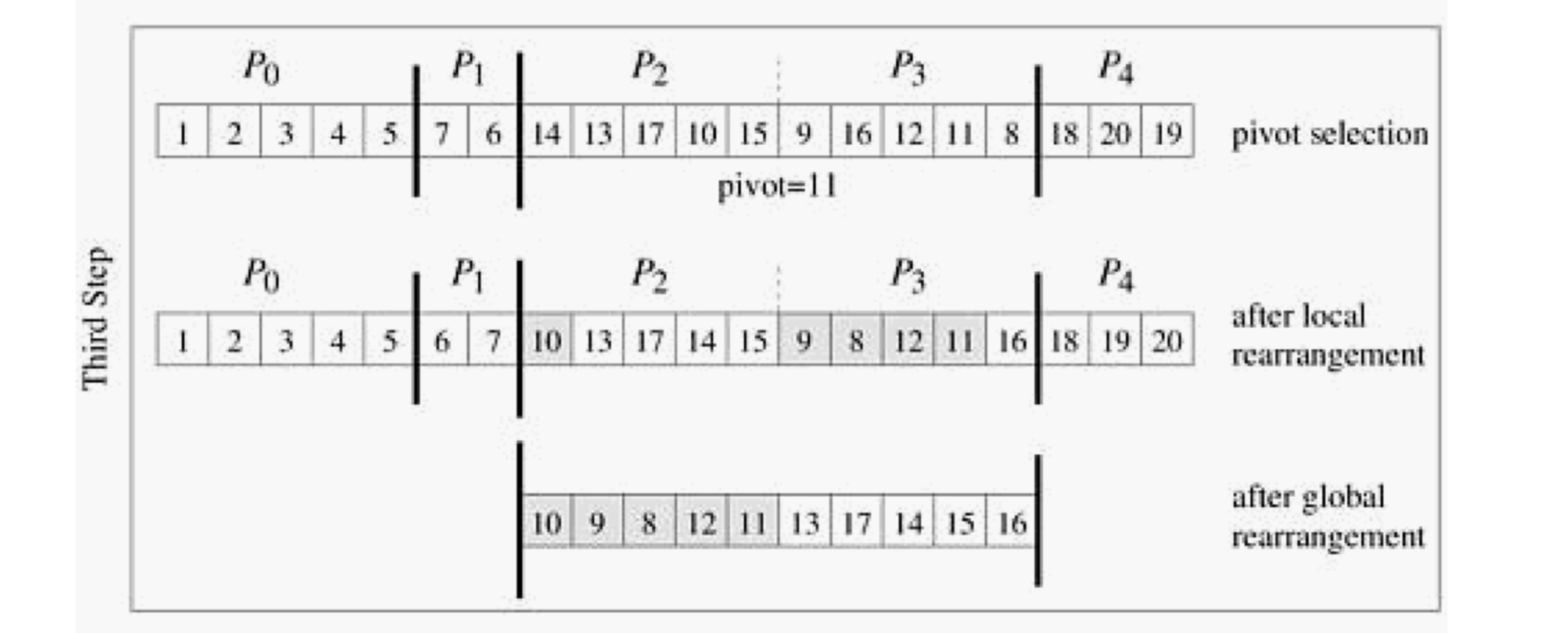
the 0.5 term in the above formula is to ensure that the processes are assigned in the most balanced fashion.

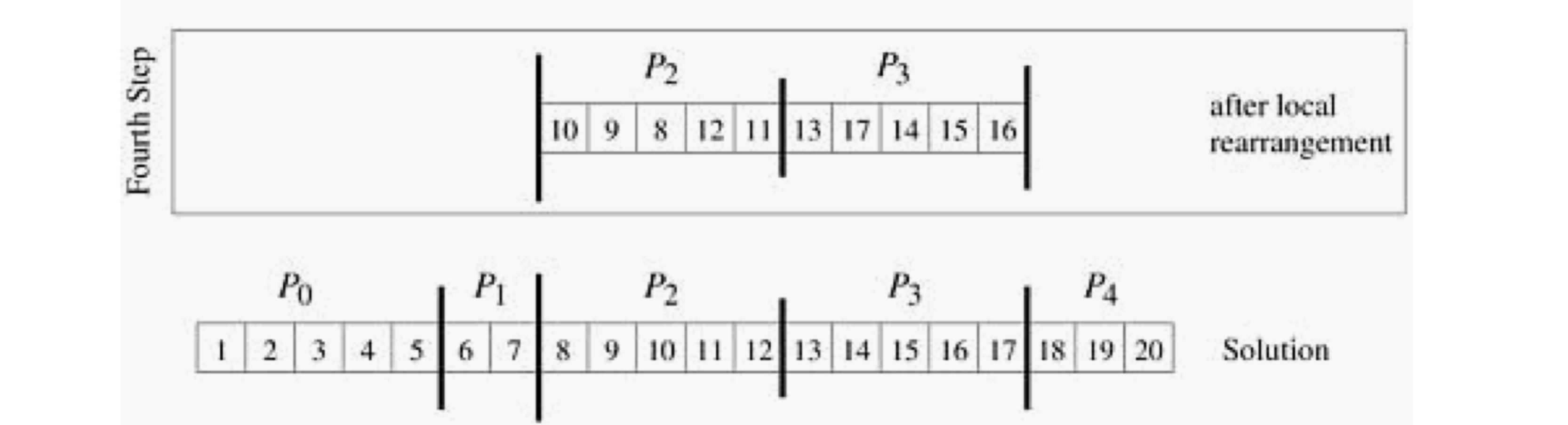
## Description of problem

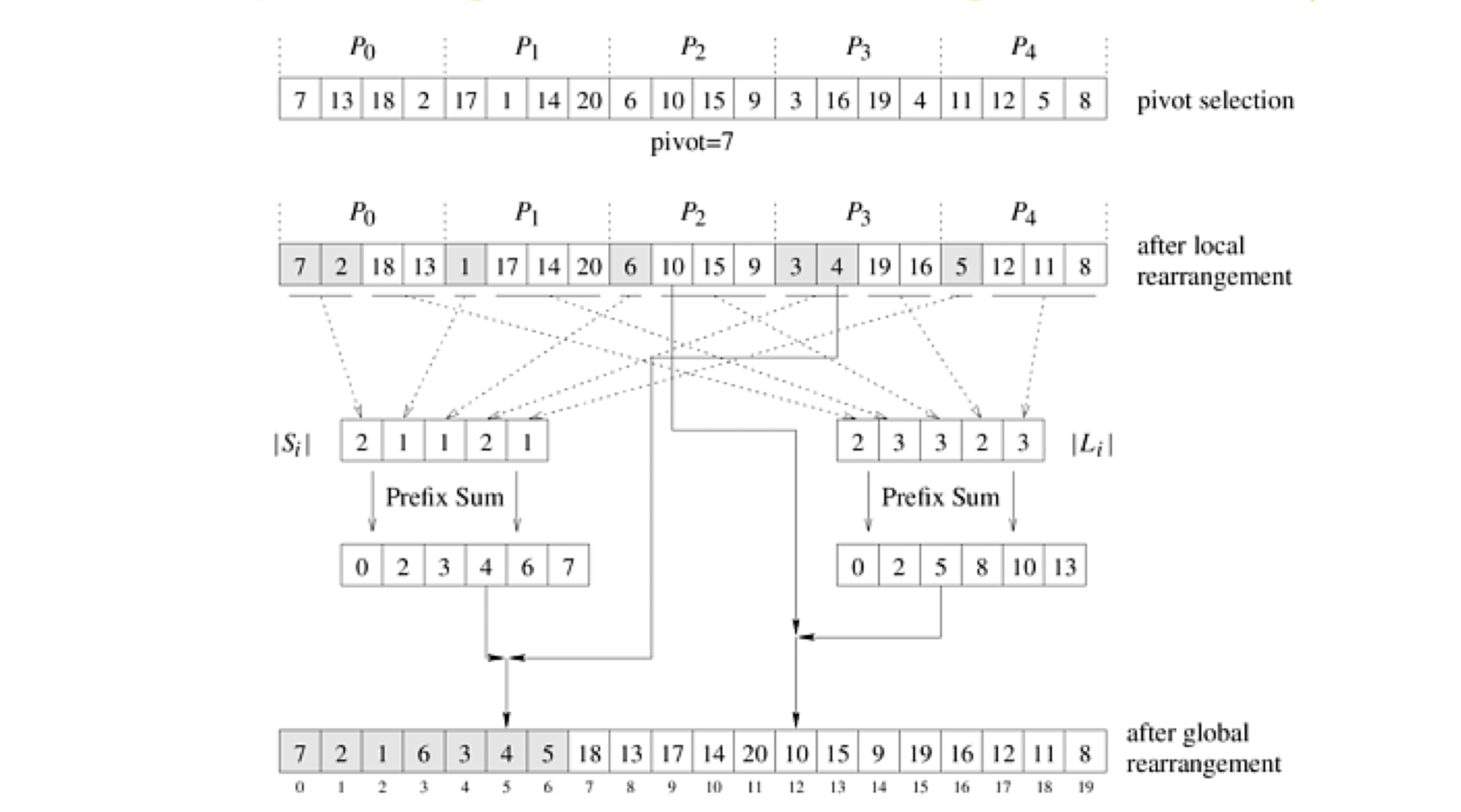
The basic algorithm is like the flowing figure.











There are some problem we have to meet.

1.The method of choosing pivot.

2.how to get prefix and split the data into two part.

3.how to use MPI\_Split, the textbook is wrong.

4.When will the loop end.

5.How to get the result.

## Contributions

### 1.The method of choosing pivot.

In parallel quick-sort system, the way of choosing pivot is different from concurrent system.

In concurrent system, we can easily choose a pivot by making the pivot equal to the first or last data in the dataset. However, in the parallel system, this can not work.

For example, for a dataset {10,9,8,7} with 2 nodes, if we choose the first data as pivot, then we will split the data set into two part {10,9,8,7} and {}. The first part will get two nodes , so it will come into a dead loop. So, we can not decide a node by simply choosing a data of dataset[0].

There are some ways to choose the pivot.

1. We can choose a random number in the dataset as the pivot. In my code, I use pivot = rand()%length;

pivot = dataset[pivot];

to find the pivot. This may also choose a terrible pivot, but in the deep iteration, it has a possibility to choose a good pivot.

1. We can find the largest and the smallest number in the dataset, then get the average as the pivot. This pivot will be a good pivot, however, this will cost much operation if the size of dataset is huge.
2. We can get the average of the dataset as the pivot. This pivot is good, but if the data is large, the sum may beyond the boundary of the integer. So this method is not good to choose.

### 2.how to get prefix and split the data into two part.

We use the prefix to split the data. However, getting the prefix\_sum is not easy. We cannot easily use MPI\_Gather() to get the small group of number because we may get different count of integer from other node. So we first gather the prefix first and then we save this prefix into a array.

MPI\_Gather(&prefix , 1, MPI\_INT,recvPrefix,1,MPI\_INT,0,comm);

Then we use the number in array to manually use MPI\_Send and MPI\_Recv to get the large group of data and the small group data. Then we broadcast this two dataset to other node.

if(node == 0)

{

prefixplus = recvPrefix[0];

memcpy(recvnum,dataset,prefix\*sizeof(int));

for(i=1;i<nodenum;i++)

{

MPI\_Recv(recvnum+prefixplus, recvPrefix[i], MPI\_INT, i,0,comm,&status);

prefixplus+=recvPrefix[i];

}

}

else

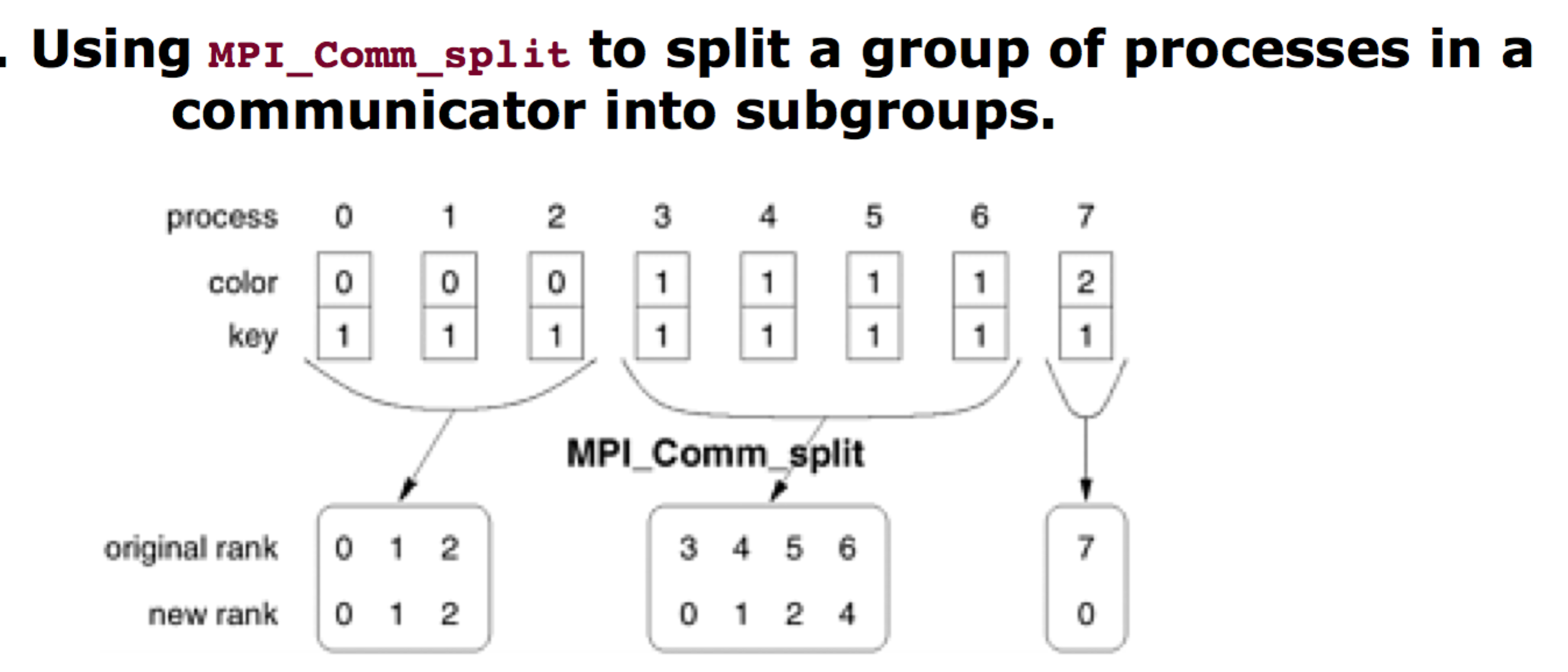
MPI\_Send(dataset+length/nodenum\*node,prefix,MPI\_INT,0,0,comm);

### 3.how to use MPI\_Split, the textbook is wrong.

We use [S\*P/n + 0.5] to determine how many nodes the smaller data will use to make the deeper iteration. (S is the prefixsum, P is node size, n is the size of dataset) I call it a integer “smallP”. If the node rank is smaller than smallP, it will participate the deeper parallel\_quicksort with smalldata, the others will participate with largedata.

smallP = prefixplus \* nodenum / length + 0.5;

Then we use int MPI\_Comm\_split(MPI\_Comm comm, int color, int key,    MPI\_Comm \*newcomm)method to split the MPI\_Comm.



this figure is on the textbook but it is wrong. If we use the same key as the parameter, it will not get what we want. So, in my code, I use following:

if(node<smallP)

MPI\_Comm\_split(orginalComm,0,node,&smallComm);

else

MPI\_Comm\_split(orginalComm,1,node,&smallComm);

We make a new MPI\_Comm. This MPI\_Comm split into 2 color. The Color 0 will manage small data and the Color 1 will manage large data. If we use MPI\_Comm\_rank, we can get the rank of node is: 0, 1, 2….. in color 0 and 0,1,2..in color 1.

So we use

if(node < smallP)

{

if(prefixplus >1)

{

parallel\_quicksort\_plus(smallDataset, prefixplus, smallPivot,smallComm);

}

}

else

{

if(prefixplus < length-1)

{

parallel\_quicksort\_plus(largeDataset, length-prefixplus,largePivot, smallComm);

}

}

to split and make the deeper iteration.

### 4.When will the loop end.

It will end in many situations.

1. The number size is 1. So it cannot split anymore. We can use concurrent quicksort to get the result.
2. smallP > node size -1. Then there are just few largedata that we split no node to handle it. So we should use concurrent quicksort for the largedata.
3. smallP =0. Then we will split no node to the smalldataset, we should use concurrent quicksort on it.
4. Prifixsum <2 or Prifixsum > length-2. If the prefixsum < 2, there are just 1 or 0 number in smalldataset, so we do not need to compare it. Prifixsum > length-2 is just the same to largedataset.

### 5.How to get the result.

We set \*dataset as the parameter of the function. So after we get the smalldataset and largedataset, we will combine them together as the new dataset. Be careful that the node with rank 0 can not get the largedataset, so we have to broadcast.

MPI\_Bcast(smallDataset,prefixplus,MPI\_INT,0,comm);

MPI\_Bcast(largeDataset,length-prefixplus,MPI\_INT,nodenum-1,comm);

memset(dataset,0,length\*sizeof(int));

memcpy(dataset,smallDataset,prefixplus\*sizeof(int));

memcpy(dataset+prefixplus,largeDataset, (length-prefixplus)\*sizeof(int));

## Evaluate

I made two datasets to Evaluate the method.

First contains 100 numbers from 100 to 1.(dataset is “inputdata.txt” and the output is in “outputtext”)

Second contains 1000000 numbers of random integers. ( dataset is “dataset.txt” and the output is in “output”)

Both of datasets we handled with 1,4,8,12,16 processors.

In test data, we can see that all of the output data is

===============

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

===============

So the correctness of parallel\_quicksort function is right.

For time:

Processor = 1: time is 0.000030 s

Processor = 4: time is 0.000653 s ,S = 0.065,E=0.01625

Processor = 8: time is 0.002283 s ,S = 0.013,E=0.01625

Processor = 12: time is 0.005903 s ,S = 0.005,E=0.00043

Processor = 16: time is 0.006915 s ,S = 0.004,E=0.00025

For the huge dataset, we can see the solution of parallel\_dataset is the same as concurrent\_dataset.

For time:

Processor = 1: time = 0.189154 s

Processor = 4: time = 0.206604 s , S = 0.9155 , E = 0.2289

Processor = 8: time = 0.461206 s , S = 0.4101 , E = 0.0513

Processor = 12: time = 0.530901 s , S = 0.3563 , E = 0.0297

Processor = 16: time = 0.568842 s , S = 0.3325 , E = 0.0208

So we can find that when the dataset is huge, the speedup and efficient of parallel\_quicksort is much better than small dataset.

## Conclusion

This project makes us get a higher understanding of parallel system. By doing this project we can learn more about the data transfer, the community split and so on. We can know that the parallel system is not in good use when dataset is small, but as the dataset become huge, the parallel system will be more and more efficient.