# 6. MPI在一些统计方法上的应用

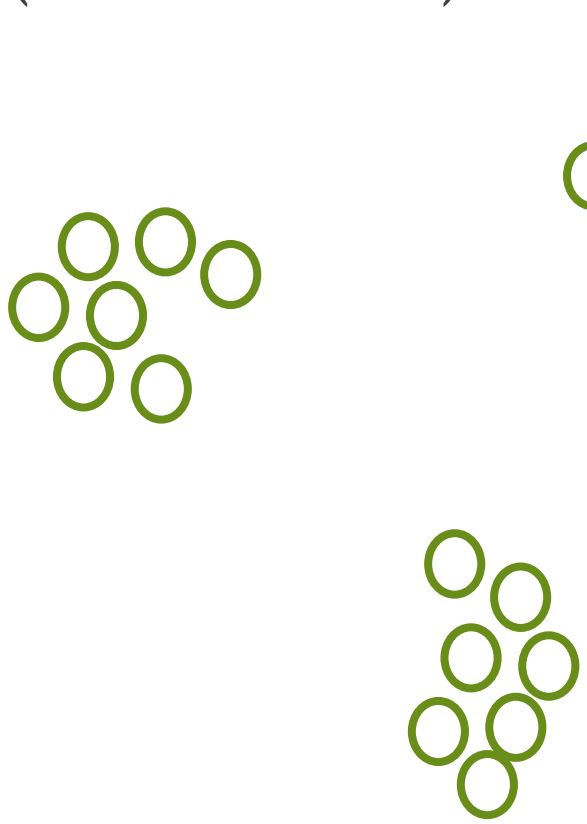
罗翔宇
中国人民大学统计与大数据研究院

# MPI学习总览

- MPI点对点通信：MPI_Send, MPI_Recv
- MPI集体通信：MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Reduce
- MPI部分通信：分组+集体通信
  - MPI_Comm_group，MPI_Group_incl, MPI_Comm_create
  - MPI_Comm_split

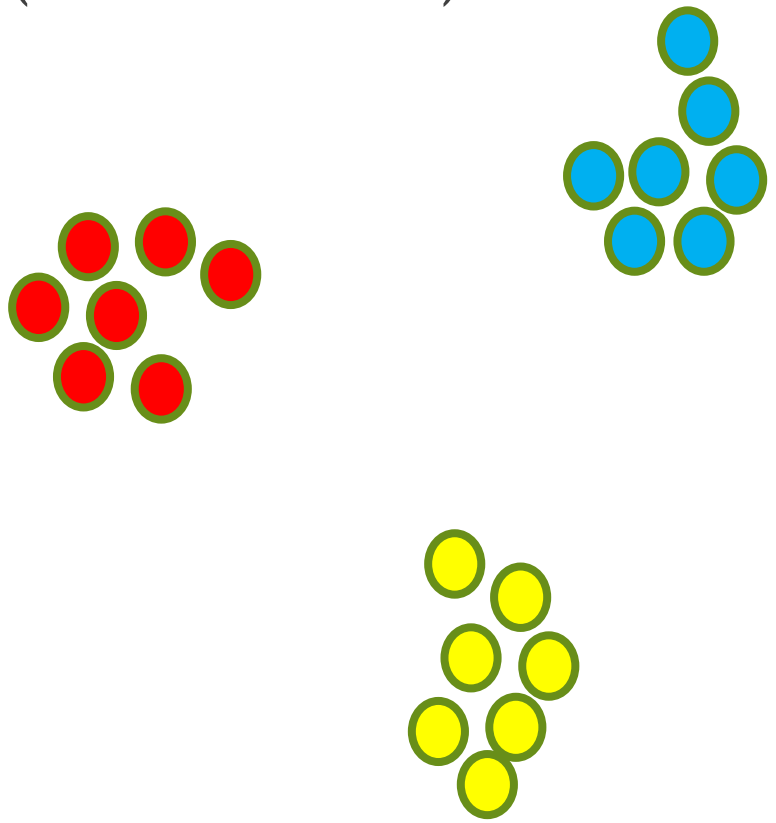
# 什么是Kmeans

- Kmeans是一种聚类(clustering)算法，将若干数据聚集成K类
- Kmeans不需要数据的标签，它是非监督(unsupervised)
  - 和分类(classification)不同



# 什么是Kmeans

- Kmeans是一种聚类(clustering)算法，将若干数据聚集成K类
- Kmeans不需要数据的标签，它是非监督(unsupervised)
  - 和分类(classification)不同



# Kmeans的数学表示

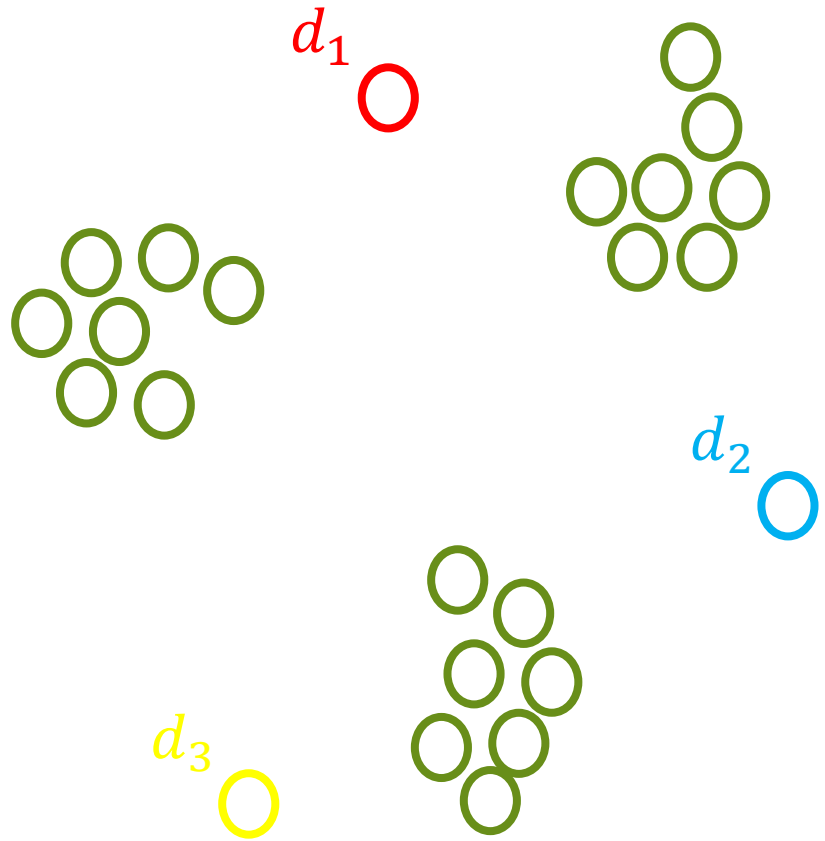
- 假设我们有N个数据点$x_1, x_2, \ldots, x_N$，计划用Kmeans将其聚为K类
- 假设数据点$x_i$来自于类型$Z_i \in \{1,2,\ldots,K\}$
- 用$C_k$表示属于类型k的数据点的角标集合，即$C_k = \{i: Z_i = k\}$
- 因此$C_k (1 \leq k \leq K)$就构成对$\{1,2,\ldots,N\}$的一个划分
- Kmeans的目的为优化如下目标函数

$$\min_{C_1,\ldots,C_K} \sum_{1\leq k\leq K} \sum_{i\in C_k} \left( x_i - \frac{1}{|C_k|} \sum_{i\in C_k} x_i \right)^2$$

- $d_k = \frac{1}{|C_k|}\sum_{i\in C_k} x_i$ 表示类k的中心点
- 但是，此问题是NP hard

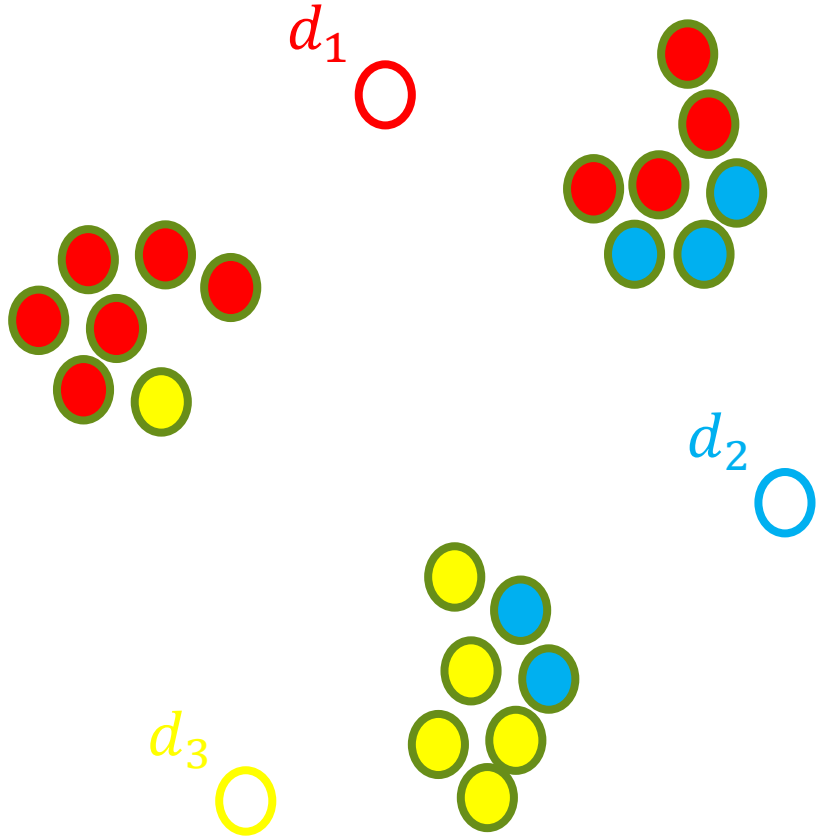
# Kmeans的迭代算法

- 迭代算法举例如下(K=3)
- 第0步，初始化中心点$d_1, d_2, d_3$



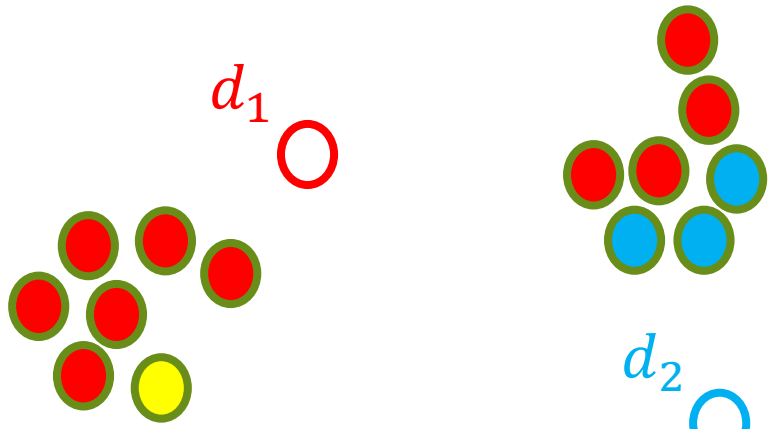
# Kmeans的迭代算法

- 第1步，计算每个点到中心点的距离，并将此点归于距离最小的那个类



# Kmeans的迭代算法

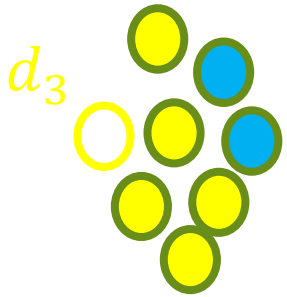
- 第2步，根据每个点所在类，重新计算每个类的中心点
- $d_k = \frac{1}{|C_k|}\sum_{i \in C_k} x_i$





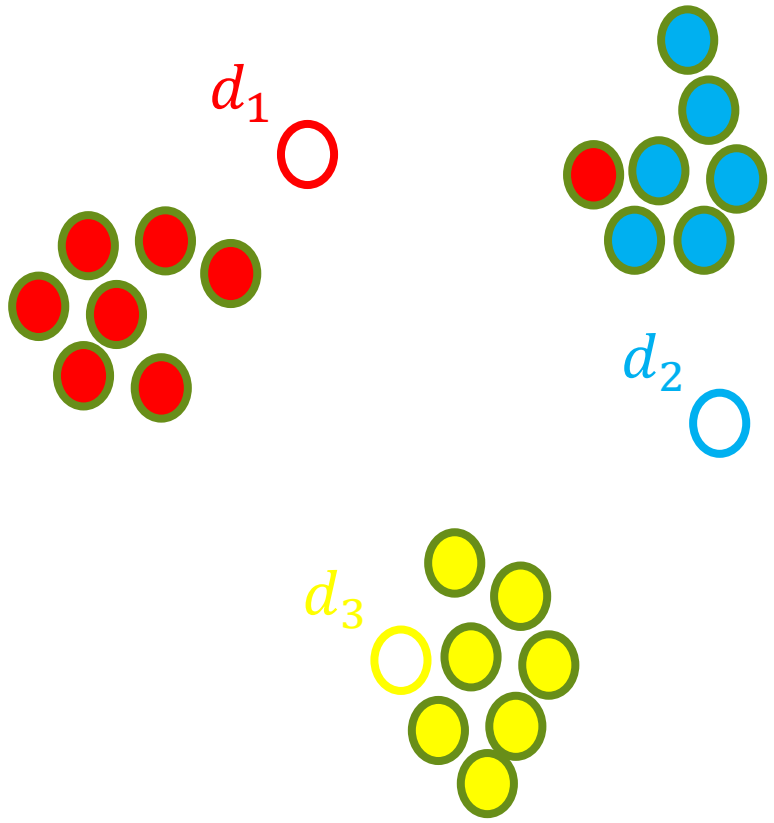






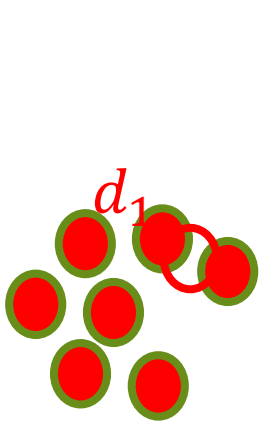
# Kmeans的迭代算法

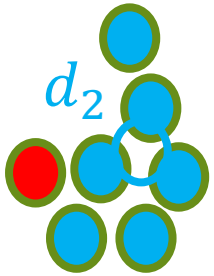
- 不断重复第1步、第2步，直到结果稳定。

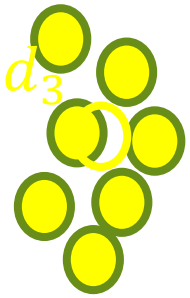


# Kmeans的迭代算法

- 不断重复第1步、第2步，直到结果稳定。

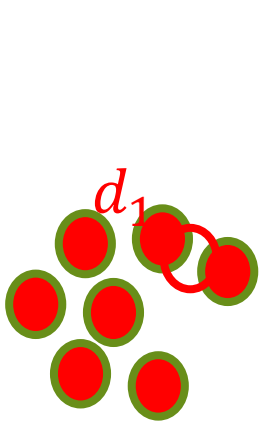


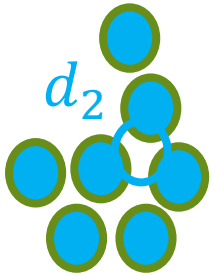



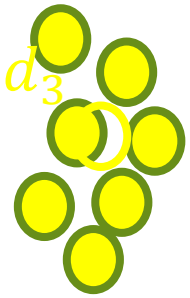



# Kmeans的迭代算法

- 不断重复第1步、第2步，直到结果稳定。

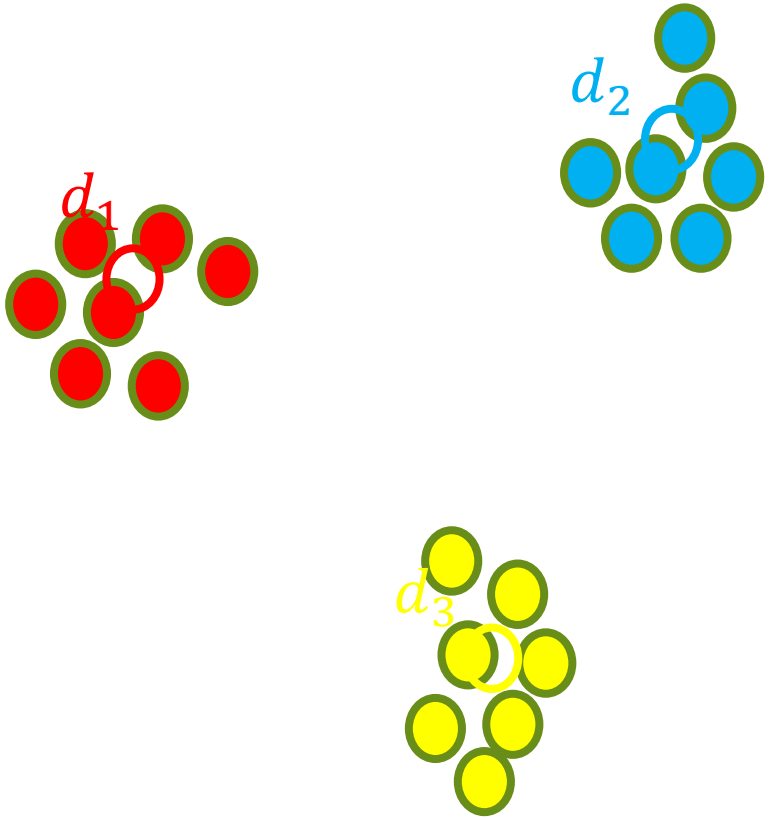


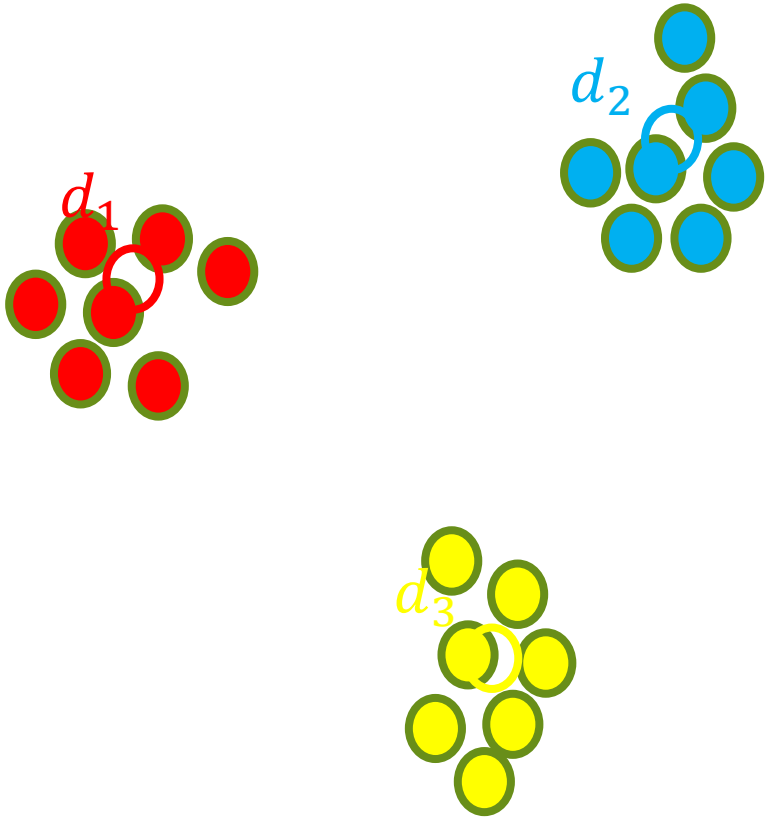
# Kmeans的迭代算法

- 不断重复第1步、第2步，直到结果稳定。



# Kmeans的迭代算法两个关键函数

- 第一步：计算$Z_i$。每个点到各个中心点的距离，找到最短的那个中心点，把自己的类型调整为最短中心点所在类型
- 第二步：计算$d_k$。根据现有每个点所在类型，更新中心点



# Kmeans的迭代算法并行化

- 当数据量很大的时候，如何提升Kmeans算法的速度呢?
- 假设我们有如下数据结构

| x1 | x2 | x3 | ... | xN-1 | xN |
|---|---|---|---|---|---|
| 0.1 | 0.2 | -0.5 | ... | 0.1 | -0.5 |
| -0.2 | -0.2 | 0.3 | ... | -0.1 | 0.25 |

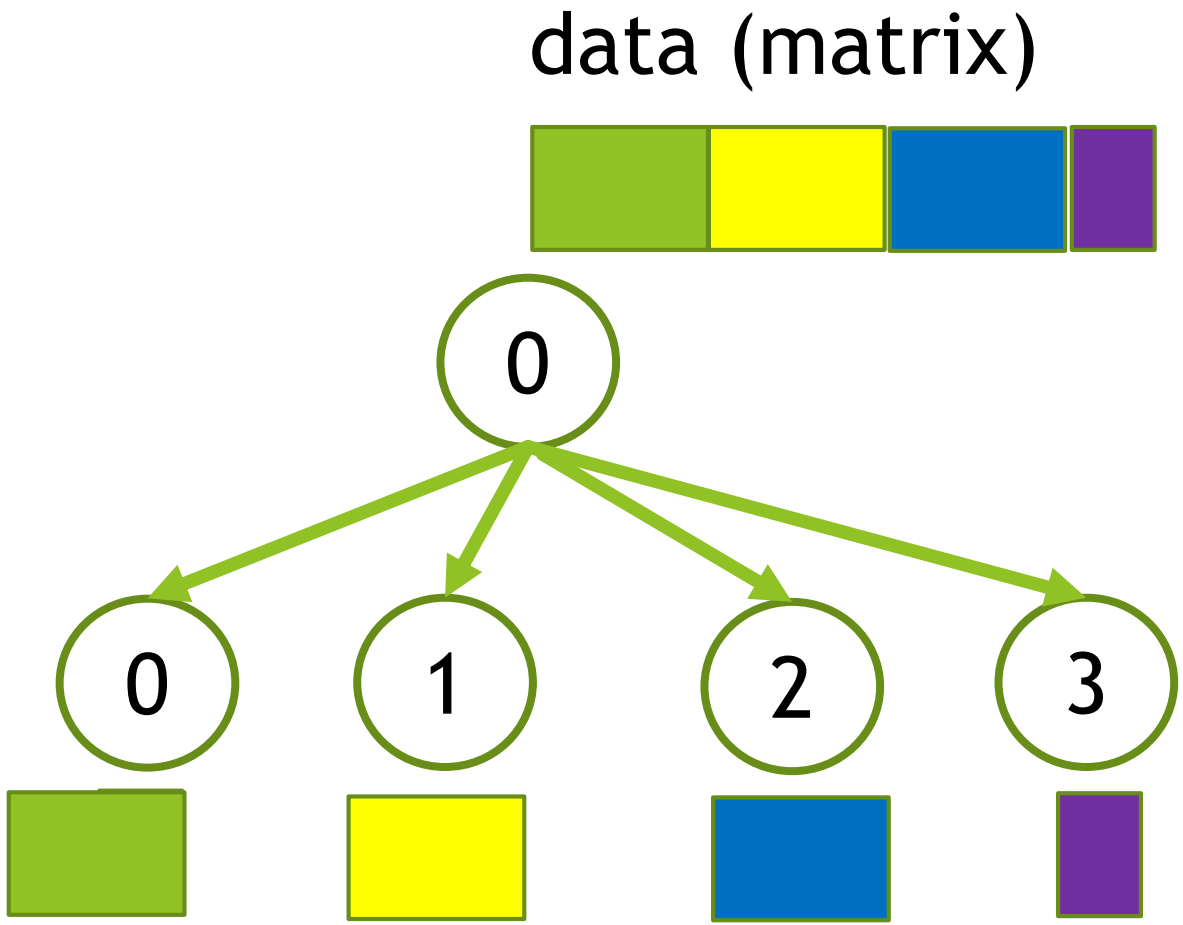
- 我们将其划分为若干部分,每一部分在对应进程上计算$Z_i$

| x1 | x2 | x3 | ... | xN-1 | xN |
|---|---|---|---|---|---|
| 0.1 | 0.2 | -0.5 | ... | 0.1 | -0.5 |
| -0.2 | -0.2 | 0.3 | ... | -0.1 | 0.25 |

- 接着将$Z_i$的值汇集到进程0上，计算$d_k$

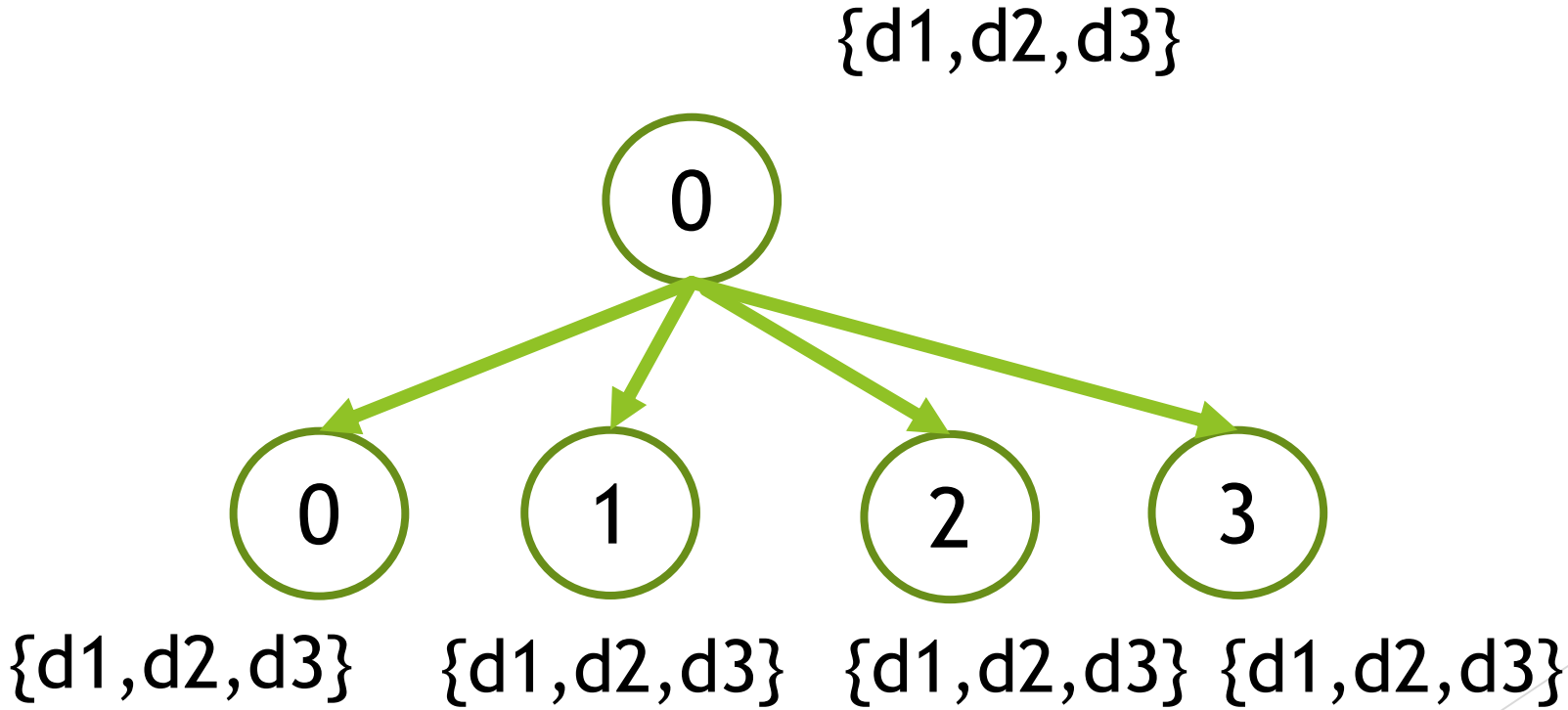
# Kmeans的迭代算法并行化

- 并行Kmeans算法示意图
- 分散数据(允许数据不均匀分散)[MPI_Scatter + MPI_Group_incl]



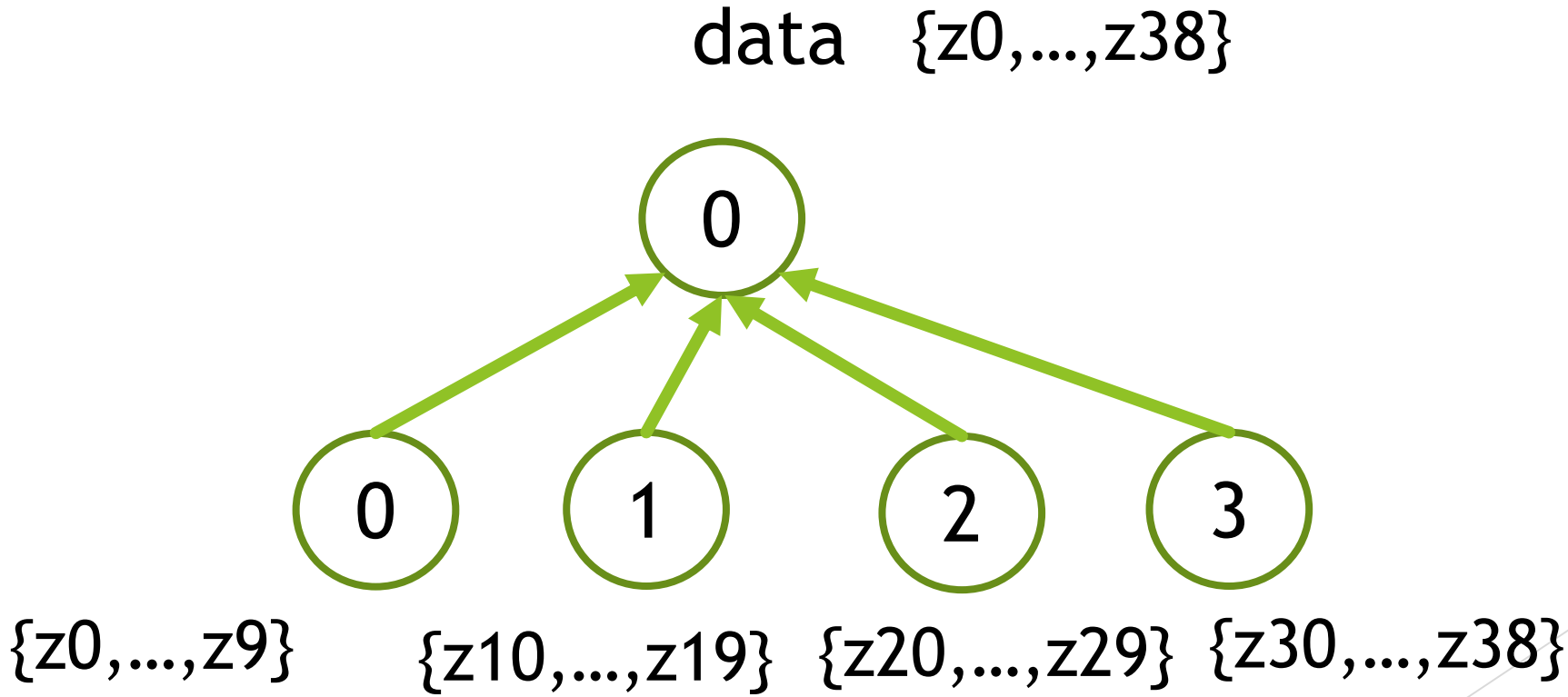
# Kmeans的迭代算法并行化

- 分散中心点
- MPI_Bcast



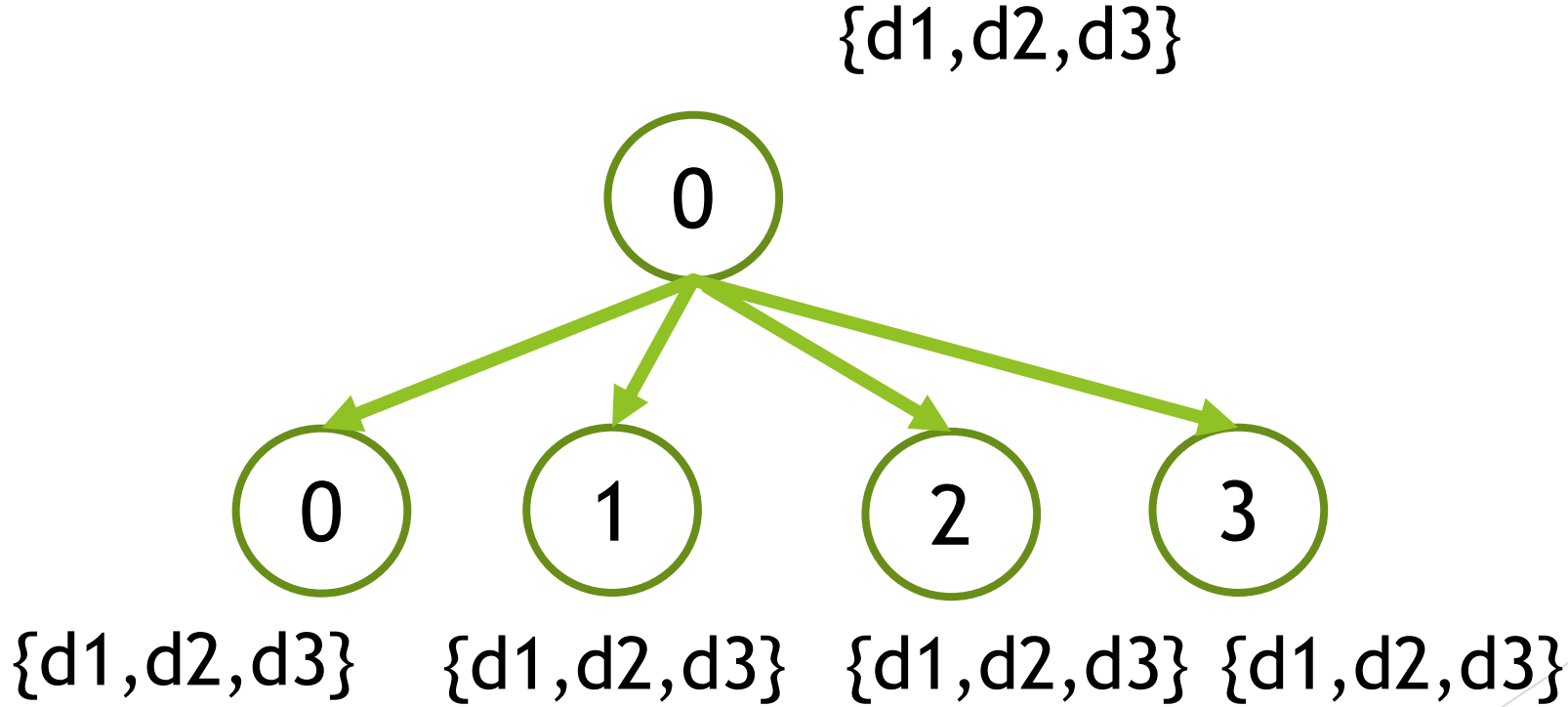
# Kmeans的迭代算法并行化

- 第一步：每个进程上并行计算$Z_i$(每个点所在类型)并汇总于进程0
- MPI_Gather



# Kmeans的迭代算法并行化

- 第二步：在进程0上更新中心点，
- 并广播到各个进程（第一次迭代结束）



# Kmeans并行化代码：

```
#include <math.h>

#define N 9999999
#define K 3
#define PI 3.141592654

//sample a random number on (0,1)
double runif(){
        double tmp;
        do{
                tmp = (double) rand() / (double) RAND_MAX;
        }while(tmp>=1 || tmp <=0);
        return tmp;
}

//sample Gaussian-distributed random numbers via Box-Muller algorithm
double rnorm(double mean, double sd){
        double tmp;
        double U, V;
        U = runif();
        V = runif();

        tmp = sqrt(-2.0*log(U)) * sin(2.0*PI*V);
        tmp = mean + sd*tmp;

        return tmp;
}
```

1千万减1个点，3类

定义从正态分布中进行抽样的函数

# Kmeans并行化代码：

```
//define a data matrix
double** make2Darray(int a1, int a2){
        double *tmp_vec = (double *)malloc(a1*a2*sizeof(double));
        double **tmp;
        tmp = (double **)malloc(a1*sizeof(double *));
        for(int i=0; i<a1; i++){
                tmp[i] = &(tmp_vec[i*a2]);
        }
        return tmp;
}

//free a data matrix
void delet2Darray(double **tmp){
        free(tmp[0]);
        free(tmp);
}
```

定义数据矩阵

# Kmeans并行化代码：

```
//compute distances between data and centers
// and return which cluster each data point belongs to

int* compute_Z(double **dat, int n_dat, int p, double **center){
        double dist, dist_tmp;
        int *Z;
        Z = (int*) malloc(n_dat * sizeof(int));

        for(int i=0; i < n_dat; i++){
                for(int k=0; k < K; k++){
                        dist_tmp = 0;
                        for(int j=0; j<p; j++){
                                dist_tmp += pow(dat[i][j] - center[k][j], 2);
                        }
                        if(k==0){
                                dist = dist_tmp;
                                Z[i] = 0;
                        }else if(dist_tmp < dist){
                                dist = dist_tmp;
                                Z[i] = k;
                        }
                }
        }

        return Z;
}
```

定义执行Kmeans第一步的函数

# Kmeans并行化代码：

```
//compute the centers
double ** compute_d(double **dat, int p, int *Z){
        double s;
        int num;
        double **d = make2Darray(K, p);
        for(int k=0; k < K; k++){
                for(int j=0; j < p; j++){
                        s=0;
                        num=0;
                        for(int i=0; i < N; i++){
                                if(Z[i] == k){
                                        s += dat[i][j];
                                        num += 1;
                                }
                        }
                        d[k][j] = s / num;
                }
        }
        return d;
}
```

定义执行Kmeans第二步的函数

# Kmeans并行化代码：

```
int main(){
        MPI_Init(NULL, NULL);
        int world_rank, world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

        //create a new communicator
        int *rank;
        rank = (int*) malloc(sizeof(int)*(world_size-1));
        for(int i=0; i < world_size-1; i++){
                rank[i] = i;
        }

        int *Z_partial;

        MPI_Group world_group, new_group;
        MPI_Comm new_comm;

        MPI_Comm_group(MPI_COMM_WORLD, &world_group);
        MPI_Group_incl(world_group, world_size-1, rank, &new_group);
        MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
```

定义新的通信器

# Kmeans并行化代码：

```
//generate data on process 0
double **dat, **dat_partial;
int p=2;
if(world_rank == 0){
        srand(123);
        dat = make2Darray(N, p); //rows are samples
        for(int i=0; i < N; i++){
                if(i < N/3){
                        dat[i][0] = rnorm(1, 0.2);
                        dat[i][1] = rnorm(1, 0.2);
                }else if(i < 2*(N/3)){
                        dat[i][0] = rnorm(1, 0.2);
                        dat[i][1] = rnorm(-1, 0.2);
                }else{
                        dat[i][0] = rnorm(-1, 0.2);
                        dat[i][1] = rnorm(-1, 0.2);

                }
        }
}
```

在进程0上生成用于聚类的数据

请注意：行对应样本，而非列对应样本，为什么？

# Kmeans并行化代码：

```
//scatter data
if(world_rank < world_size - 1){
        dat_partial = make2Darray(N/world_size, p);
}else{
        dat_partial = make2Darray(N - (world_size-1)*(N/world_size),
                        p);
}

if(new_comm != MPI_COMM_NULL){
        MPI_Scatter(dat[0], p*(N/world_size), MPI_DOUBLE,
                        dat_partial[0], p*(N/world_size), MPI_DOUBLE,
                        0, new_comm);
}

if(world_rank == 0){
        MPI_Send(dat[(world_size-1)*(N/world_size)],
                        p*(N - (world_size-1)*(N/world_size)),
                        MPI_DOUBLE,
                        world_size-1, 888, MPI_COMM_WORLD);
}


if(world_rank == world_size -1){
        MPI_Recv(dat_partial[0], p*(N - (world_size-1)*(N/world_size)),
                        MPI_DOUBLE, 0, 888,
                        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

将数据分散出去

# Kmeans并行化代码：

```
//initialize centers on process 0
double **d;
d = make2Darray(K, p);

if(world_rank == 0){
        d[0][0] = 0;
        d[0][1] = 0;

        d[1][0] = 1;
        d[1][1] = -3;

        d[2][0] = -1;
        d[2][1] = -3;
        printf("==================Initialization=================\n");
        printf("cluster 1's center is (%f, %f)\n", d[0][0], d[0][1]);
        printf("cluster 2's center is (%f, %f)\n", d[1][0], d[1][1]);
        printf("cluster 3's center is (%f, %f)\n", d[2][0], d[2][1]);
}



MPI_Bcast(d[0], K*p, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

初始化中心点

# Kmeans并行化代码：

```
//Kmeans
int t, *Z;

if(world_rank == 0){
        Z = (int*)malloc(N*sizeof(int));
}

double t1, t2;

MPI_Barrier(MPI_COMM_WORLD);
if(world_rank == 0){
        t1 = MPI_Wtime();
}
for(t=0; t < 5; t++){
        //step 1: obtain the cluster indicators
        if(world_rank < world_size - 1){
                Z_partial = compute_Z(dat_partial, N/world_size, p, d);
        }else{
                Z_partial = compute_Z(dat_partial,
                                N - (world_size-1)*(N/world_size),
                                p, d);
        }
        MPI_Barrier(MPI_COMM_WORLD);
```

执行Kmeans

# Kmeans并行化代码：

```
//gathering
if(new_comm != MPI_COMM_NULL){
        MPI_Gather(Z_partial, N/world_size, MPI_INT, Z,
                        N/world_size, MPI_INT, 0, new_comm);
}

if(world_rank == world_size-1){
        MPI_Send(Z_partial, N - (world_size-1)*(N/world_size),
                        MPI_INT, 0, 888, MPI_COMM_WORLD);
}

if(world_rank == 0){
        MPI_Recv(&(Z[(world_size-1)*(N/world_size)]),
                        N - (world_size-1)*(N/world_size),
                MPI_INT, world_size-1, 888, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
}
```

执行Kmeans

# Kmeans并行化代码：

```
                //step2: compute centers on processor 0
                if(world_rank == 0){
                        d = compute_d(dat, p, Z);
                        printf("================== Iteration %d ===============
==\n", t);
                        printf("cluster 1's center is (%f, %f)\n", d[0][0], d[0
][1]);
                        printf("cluster 2's center is (%f, %f)\n", d[1][0], d[1
][1]);
                        printf("cluster 3's center is (%f, %f)\n", d[2][0], d[2
][1]);
                }
                MPI_Bcast(d[0], K*p, MPI_DOUBLE, 0, MPI_COMM_WORLD);
                MPI_Barrier(MPI_COMM_WORLD);

        }

        if(world_rank == 0){
                t2 = MPI_Wtime();
                printf("\nTime cost: %f seconds \n", t2 - t1);
        }


        MPI_Finalize();
}
```

执行Kmeans

# 运行结果：利用两个核

```
[luoxiangyu@node-1-14 parallel_computing]$ mpirun -np 2 ./Kmeans
==================Initialization==================
cluster 1's center is (0.000000, 0.000000)
cluster 2's center is (1.000000, -3.000000)
cluster 3's center is (-1.000000, -3.000000)
================== Iteration 0 ==================
cluster 1's center is (0.346462, -0.291954)
cluster 2's center is (1.127639, -1.380798)
cluster 3's center is (-1.126114, -1.380772)
================== Iteration 1 ==================
cluster 1's center is (0.982360, 0.878207)
cluster 2's center is (1.018051, -1.025161)
cluster 3's center is (-1.000923, -1.000578)
================== Iteration 2 ==================
cluster 1's center is (1.000094, 1.000196)
cluster 2's center is (1.000029, -1.000020)
cluster 3's center is (-1.000063, -0.999942)
================== Iteration 3 ==================
cluster 1's center is (1.000094, 1.000199)
cluster 2's center is (1.000028, -1.000018)
cluster 3's center is (-1.000063, -0.999942)
================== Iteration 4 ==================
cluster 1's center is (1.000094, 1.000199)
cluster 2's center is (1.000028, -1.000018)
cluster 3's center is (-1.000063, -0.999942)

Time cost: 1.984286 seconds
```

# 运行结果：利用四个核

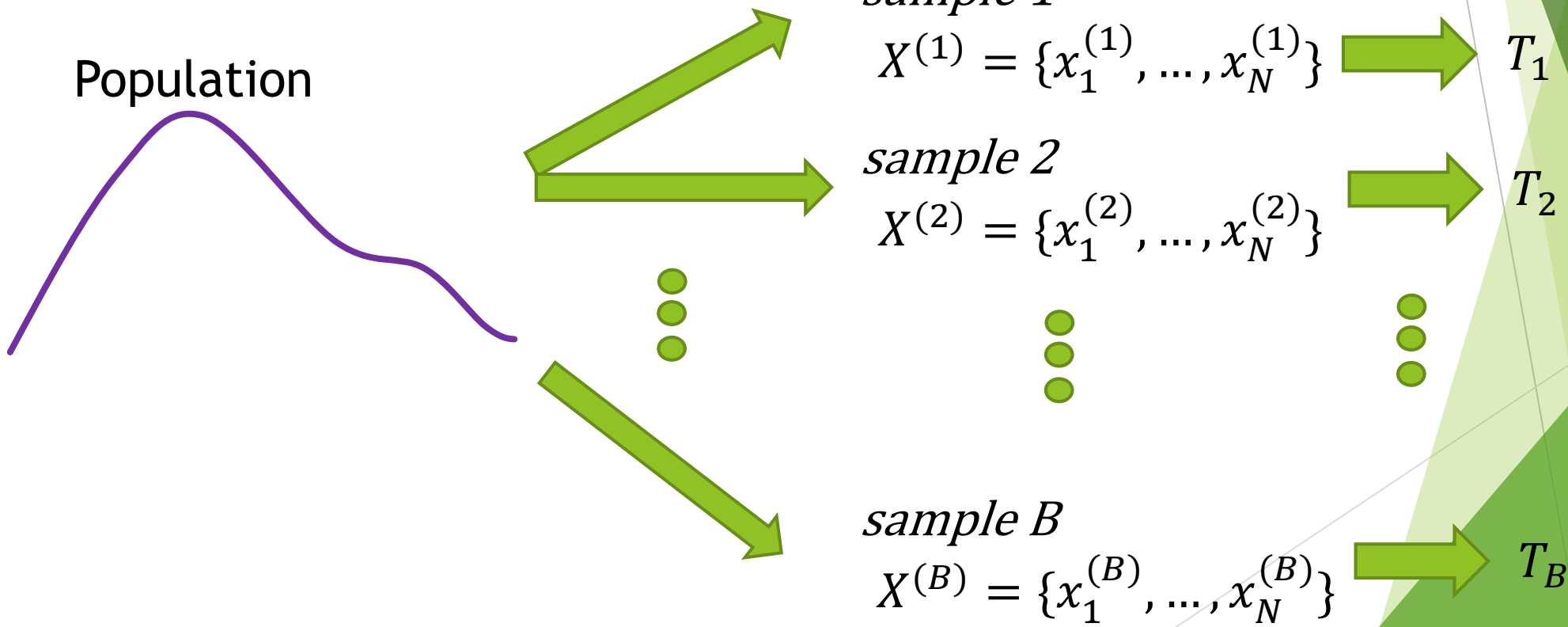
```
[luoxiangyu@node-1-14 parallel_computing]$ mpirun -np 4 ./Kmeans
==================Initialization==================
cluster 1's center is (0.000000, 0.000000)
cluster 2's center is (1.000000, -3.000000)
cluster 3's center is (-1.000000, -3.000000)
================== Iteration 0 ==================
cluster 1's center is (0.346462, -0.291954)
cluster 2's center is (1.127639, -1.380798)
cluster 3's center is (-1.126114, -1.380772)
================== Iteration 1 ==================
cluster 1's center is (0.982360, 0.878207)
cluster 2's center is (1.018051, -1.025161)
cluster 3's center is (-1.000923, -1.000578)
================== Iteration 2 ==================
cluster 1's center is (1.000094, 1.000196)
cluster 2's center is (1.000029, -1.000020)
cluster 3's center is (-1.000063, -0.999942)
================== Iteration 3 ==================
cluster 1's center is (1.000094, 1.000199)
cluster 2's center is (1.000028, -1.000018)
cluster 3's center is (-1.000063, -0.999942)
================== Iteration 4 ==================
cluster 1's center is (1.000094, 1.000199)
cluster 2's center is (1.000028, -1.000018)
cluster 3's center is (-1.000063, -0.999942)

Time cost: 1.567688 seconds
```

# 运行结果

- 2个核：1.984秒
- 4个核：1.568秒
- 为什么速度不是一半？
  - 每次迭代，都需要进程0与其他进程进行通信(Gather, Scatter,)
  - 电脑也被其他程序所占用
  - 数据不能均匀等分，有一个额外的点对点通信
  - …

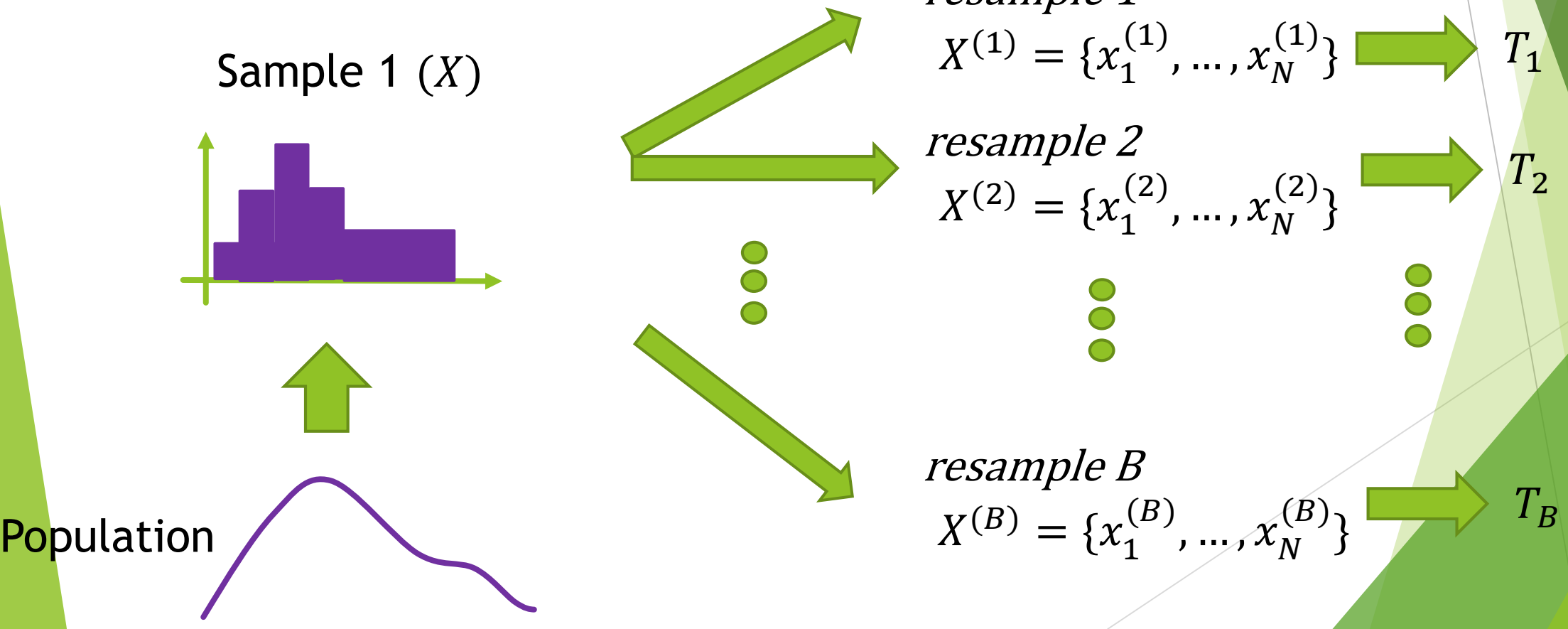
# Bootstrap

- 假设我们有数据$X = \{x_1, x_2, \ldots, x_N\}$, Bootstrap的目的是刻画出某个依赖于$X$的统计量$\mathrm{T}(X)$的分布
- 理想情形：



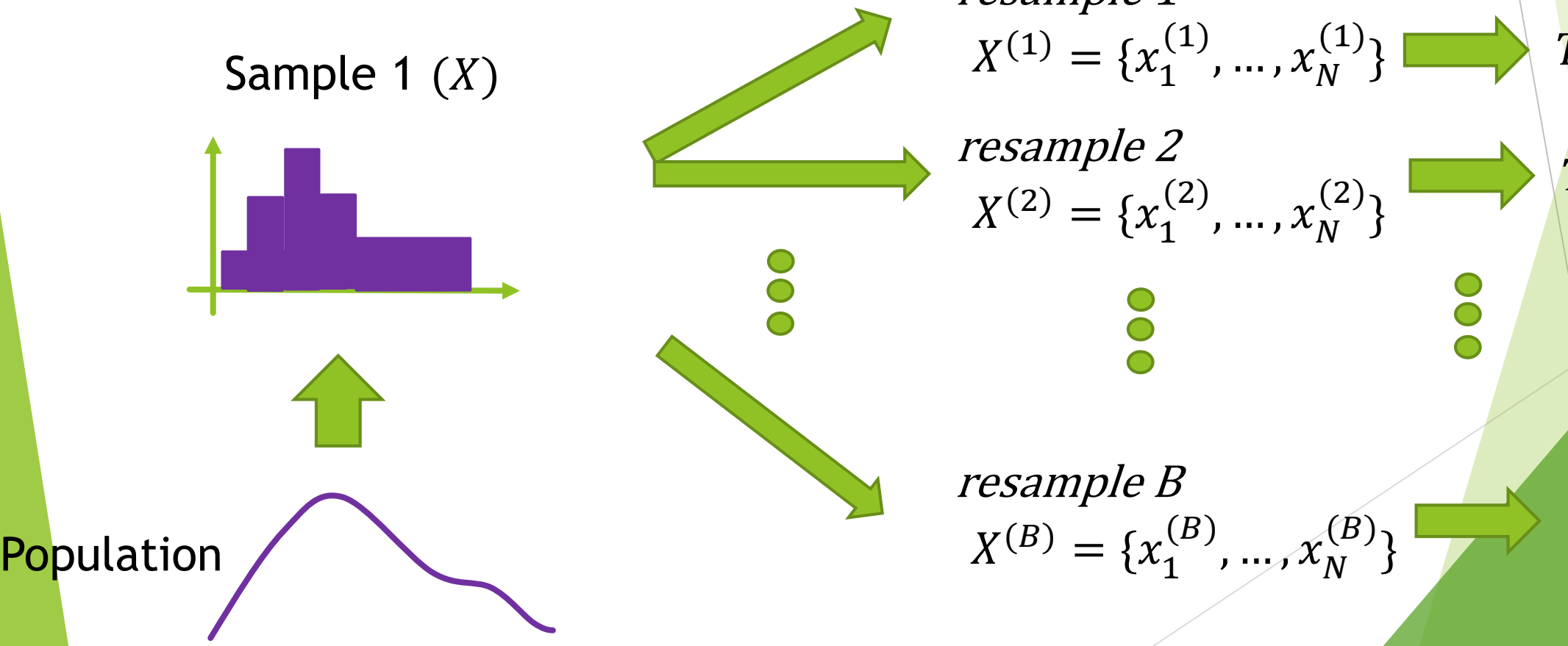
# Bootstrap

- 假设我们有数据$X = \{x_1, x_2, \ldots, x_N\}$, Bootstrap的目的是刻画出某个依赖于$X$的统计量$\mathrm{T}(X)$的分布
- 实际情形(Bootstrap)：



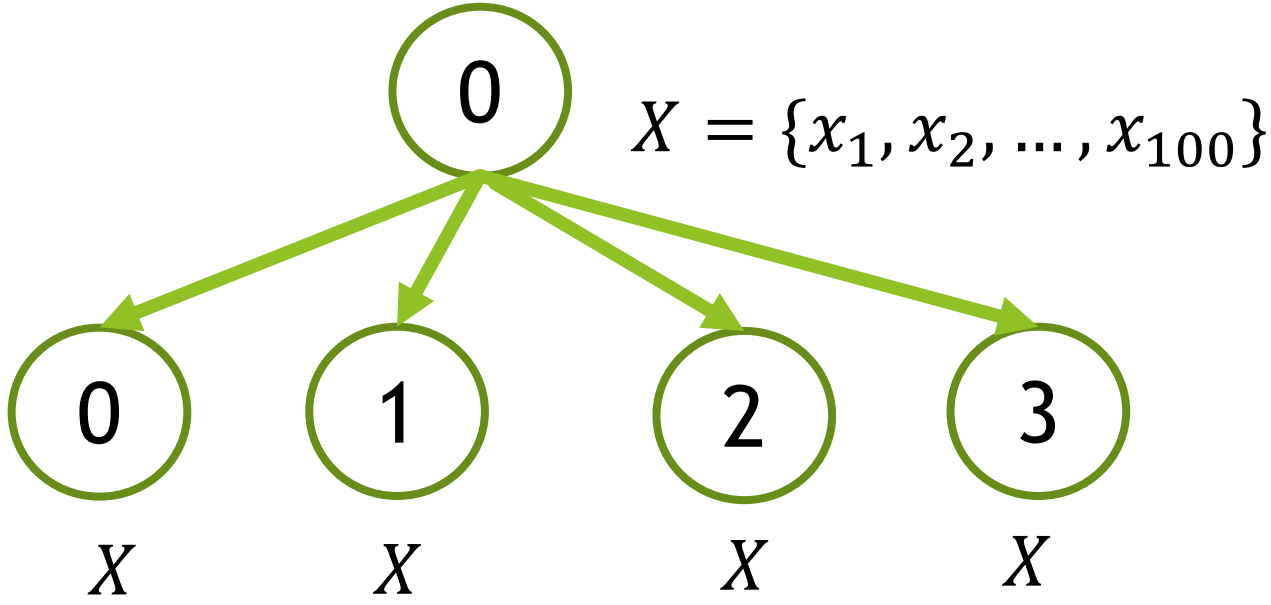
# Bootstrap

- 重抽样的过程是有放回地抽取，每次之间相互独立
- 实际情形(Bootstrap)：



# Bootstrap并行化

- 例子： $X = \{x_1, x_2, \ldots, x_{100}\}$来自于标准正态分布，通过Bootstrap求其样本方差的均值($E(S^2)$)的估计值
- 重抽样次数选为B=10万
- 调用S个进程，每个进程做重抽样B/S次



# Bootstrap并行化

```
#include <math.h>

#define B 100000
#define N 100
#define PI 3.141592654

//sample a random number on (0,1)
double runif(){
        double tmp;
        do{
                tmp = (double) rand() / (double) RAND_MAX;
        }while(tmp>=1 || tmp <=0);
        return tmp;
}

//sample Gaussian-distributed random numbers via Box-Muller algorithm
double rnorm(double mean, double sd){
        double tmp;
        double U, V;
        U = runif();
        V = runif();

        tmp = sqrt(-2.0*log(U)) * sin(2.0*PI*V);
        tmp = mean + sd*tmp;

        return tmp;
}
```

定义抽取随机数函数

# Bootstrap并行化

```
//sum function
double sum(double* array, int length){
        double s = 0;
        for(int i=0; i<length; i++){
                s += array[i];
        }
        return s;
}

//var function
double var(double* array, int length){
        double avg;
        avg = sum(array, length) / length;
        double s = 0;
        for(int i=0; i<length; i++){
                s += (array[i]-avg)*(array[i]-avg);
        }
        s = s/(length - 1);
        return s;
}
```

求方差函数

# Bootstrap并行化

```
//uniformly sample from 0,1,...,n-2,n-1
int sample(int n){
        double U = runif();
        int tmp;
        for(int i=0; i<n; i++){
                if((U > i/(double)n)&&(U < (i+1)/(double)n)){
                        tmp = i;
                        break;
                }
        }
        return tmp;
}


int main(){
        MPI_Init(NULL, NULL);
        int world_rank, world_size;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

        double *dat;
        dat = (double *)malloc(sizeof(double)*N);
```

随机抽取0至n-1中整数的函数

# Bootstrap并行化

```
if(world_rank == 0){
        srand(123);
        for(int i=0; i<N; i++)
                dat[i] = rnorm(0,1);
}

MPI_Bcast(dat, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

int B0 = B / world_size;

//Bootstrap
int t;
double s=0;
double* re_dat = (double *)malloc(sizeof(double)*N);
int index;

double t1, t2;
MPI_Barrier(MPI_COMM_WORLD);
if(world_rank == 0){
        t1 = MPI_Wtime();
}
```

定义并广播数组

# Bootstrap并行化

```
        for(t=0; t<B0; t++){
                for(int i=0; i<N; i++){
                        index = sample(N);
                        re_dat[i] = dat[index];
                }
                s += var(re_dat, N);
        }

        MPI_Barrier(MPI_COMM_WORLD);
        if(world_rank == 0){
                t2 = MPI_Wtime();
        }

        double avg;
        MPI_Reduce(&s, &avg, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

        if(world_rank == 0){
                avg = avg / B;
                printf("The bootstrap estimate of the mean of the sample varian
ce is %f\n", avg);
                printf("Time cost %f seconds\n", t2 - t1);
        }

        MPI_Finalize();
}
```

进行Bootstrap

# Bootstrap并行化结果

```
[luoxiangyu@node-3-2 parallel_computing]$ mpirun -np 2 ./Bootstrap
The bootstrap estimate of the mean of the sample variance is 1.207092
Time cost 4.004207 seconds
[luoxiangyu@node-3-2 parallel_computing]$ mpirun -np 4 ./Bootstrap
The bootstrap estimate of the mean of the sample variance is 1.207495
Time cost 2.002266 seconds
[luoxiangyu@node-3-2 parallel_computing]$ mpirun -np 8 ./Bootstrap
The bootstrap estimate of the mean of the sample variance is 1.205771
Time cost 1.001832 seconds
```

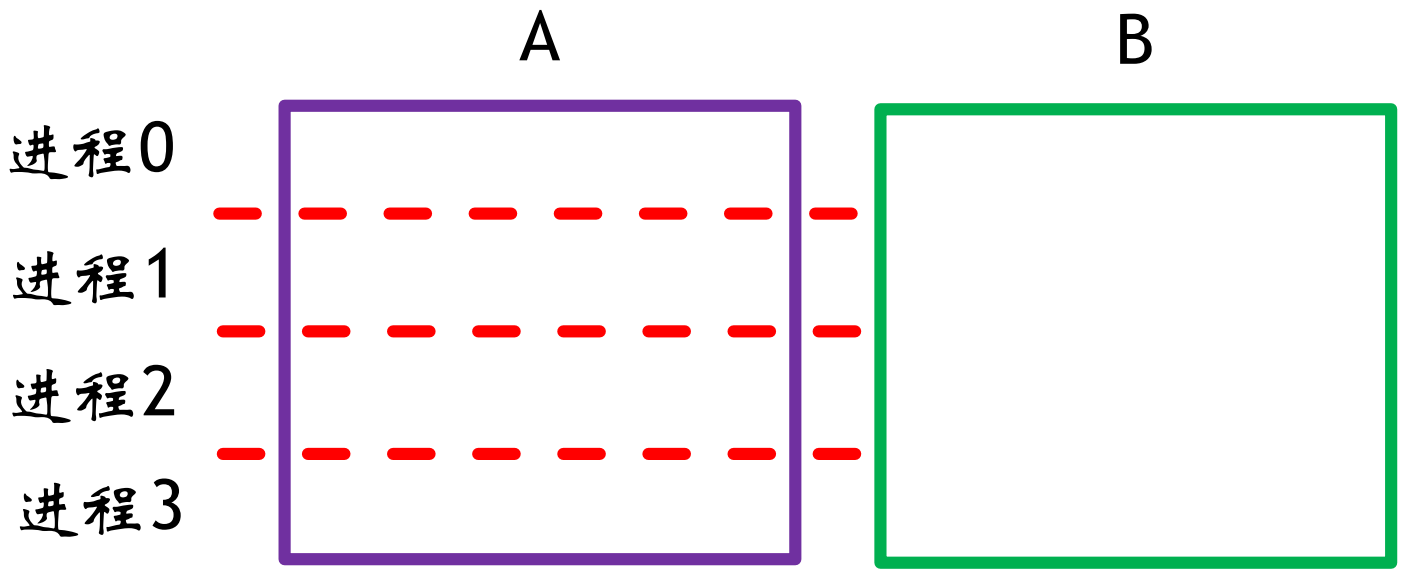
可以看到Bootstrap并行中除了开始和结束，

中间过程不需要任何通信开销，

因此增加一倍的进程(核数允许的情况下)，时间缩小一倍

# 矩阵乘法的并行化

- 矩阵乘法，假设有两个方阵A和B，想计算A和B的矩阵乘积




$$(AB)_{ij} = \sum_{1 \le k \le n} A_{ik} B_{kj}$$