4. MPI集体通信

罗翔宇
中国人民大学统计与大数据研究院

此课件内容主要基于Blaise Barney的网络资料Message Passing Interface (MPI)及Wes Kendall的MPI Tutorial

上周课堂复习

- MPI_Send中传输的数据量要小于或等于MPI_Recv中声明的接收数据量;否则，会报错。
- 进程0：MPI_Send(buf, send_count, ...)
- 进程1：MPI_Recv(buf, recv_count, ...)
- send_count <= recv_count
- 当send_count < recv_count的时候，进程1实际只接收了send_count这么多的数据量

上周课堂复习

- 动态的消息传递：在接收消息之前，可以利用MPI_Probe和MPI_Get_count来查询消息的大小
- int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status* status)
- int MPI_Get_count(MPI_Status* status, MPI_Datatype datatype, int* count)

上周课堂复习

- 动态的消息传递：可整理为以下函数
- #define MYTYPE int
- MYTYPE* MPI_Recv_dynamic (MPI_Datatype datatype, int source,
  int tag, MPI_Comm comm, int* count_number){

```
    MPI_Status stat;
    MPI_Probe(source, tag, comm, &stat);
    MPI_Get_count(&stat, datatype, count_number);
    MYTYPE *buf;
    buf= (MYTYPE *) malloc(sizeof (MYTYPE) * count_number[0]);
    MPI_Recv(buf , count_number[0], datatype, source, tag, comm,
                MPI_STATUS_IGNORE);
     return buf;
}
```

上周课堂复习

- 进程0传输随机个整数

```
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

int number_amount, i;

if(world_rank == 0){
        const int MAX_NUMBERS = 100;
        int numbers[MAX_NUMBERS];
        srand(time(NULL));
        number_amount = (rand() / (float)RAND_MAX) * MAX_NUMBERS;

        for(i=0; i<number_amount; i++){
                numbers[i] = i+1;
        }

        MPI_Send(numbers, number_amount, MPI_INT, 1, 888,
                        MPI_COMM_WORLD);
        printf("Process 0 sent %d numbers to process 1\n",
                        number_amount);
}else{
```

上周课堂复习

- 进程1动态接收

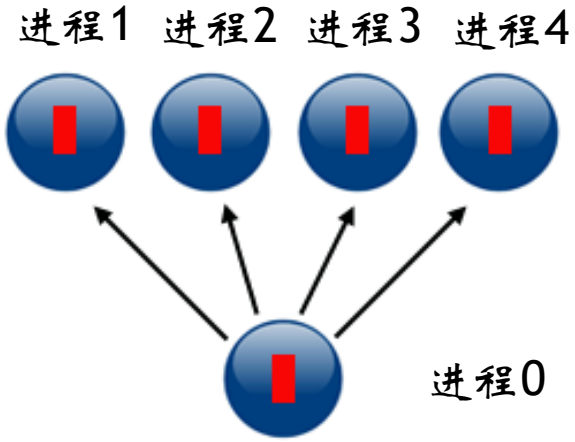
```
                int* number_buff;
                int count_num;
                number_buff = MPI_Recv_dynamic( MPI_INT, 0, 888,
                        MPI_COMM_WORLD, &count_num);
                printf("Process 1 received %d numbers from process 0\n",
                                count_num);
                for(i=0; i<count_num; i++){
                        printf("%d ", number_buff[i]);
                }
                printf("\n");
        }

        MPI_Finalize();
}
```

上周课堂复习

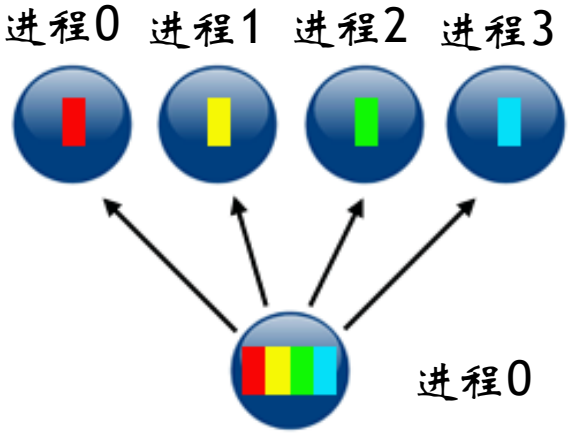
- 输出结果

```
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files/hw3$ mpicc -o receive_dyn
amic receive_dynamic.c
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files/hw3$ mpirun -np 2 ./recei
ve_dynamic
Process 0 sent 57 numbers to process 1
Process 1 received 57 numbers from process 0
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 3
0 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
```

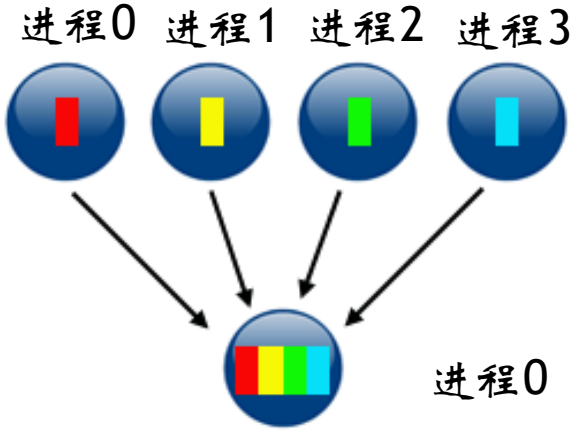
上周课堂复习



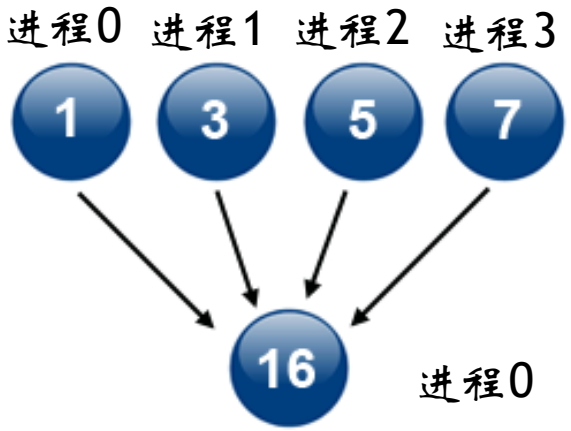
broadcast



scatter



gather

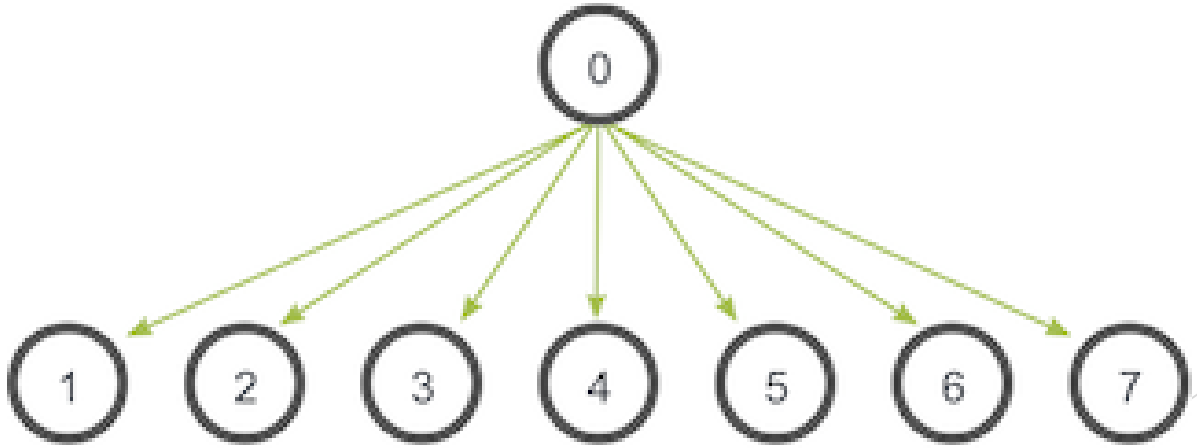


reduction

Figure credit: https://computing.llnl.gov/tutorials/mpi/

上周课堂复习

- MPI广播(broadcast)是将一个进程的数据传播到其他各个进程
- MPI内置函数 int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
  - 参数root指的是发送消息的进程rank
  - 所有进程共用此函数！



上周课堂复习

- MPI广播通信也可以用MPI点对点通信来自定义
- 当进程数量足够大时，自定义广播的速度是比MPI_Bcast慢得多的

MPI自定义广播通信与MPI_Bcast速度比较

```
int main(int argc, char** argv){
        if(argc != 3){
                fprintf(stderr, "Usage: compare_bcast num_elements num_trials\n
");
                exit(1);
        }

        int num_elements = atoi(argv[1]);
        int num_trials = atoi(argv[2]);
```

- 执行时，除了执行文件名，还需要输入两个额外参数，分别指定好每次试验的数据量(num_elements)，以及试验的数量(num_trials)

MPI自定义广播通信与MPI_Bcast速度比较

- 计算时间： #include <assert.h>

```
double total_my_bcast_time = 0.0;
double total_mpi_bcast_time = 0.0;

int i;
int* data = (int*) malloc(sizeof(int)*num_elements);
assert(data != NULL);

if(world_rank == 0){
        for(i=0; i<num_elements; i++){
                data[i] = i*i;
        }
}
```

MPI自定义广播通信与MPI_Bcast速度比较

- 计算时间：

```
for(i = 0; i < num_trials; i++){
        MPI_Barrier(MPI_COMM_WORLD);
        total_my_bcast_time -= MPI_Wtime();

        my_bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);

        MPI_Barrier(MPI_COMM_WORLD);
        total_my_bcast_time += MPI_Wtime();

        MPI_Barrier(MPI_COMM_WORLD);
        total_mpi_bcast_time -= MPI_Wtime();

        MPI_Bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);

        MPI_Barrier(MPI_COMM_WORLD);
        total_mpi_bcast_time += MPI_Wtime();
}
```

MPI自定义广播通信与MPI_Bcast速度比较

- 计算每次试验所花平均时间

```
    if(world_rank == 0){
            printf("Data size = %d, Trials = %d\n", num_elements *
                    (int)sizeof(int), num_trials);
            printf("Avg my_bcast time = %lf\n", total_my_bcast_time /
                    num_trials);
            printf("Avg MPI_Bcast time = %lf\n", total_mpi_bcast_time /
                    num_trials);
    }

    free(data);
```

MPI自定义广播通信与MPI_Bcast速度比较

- 在双核计算机上试验的结果

```
mackies-MacBook-Pro:examp mackie$ mpirun -np 2 ./compare_bcast 1000000 10
Data size = 4000000, Trials = 10
Avg my_bcast time = 0.000754
Avg MPI_Bcast time = 0.000485
mackies-MacBook-Pro:examp mackie$ mpirun -np 2 ./compare_bcast 100 10
Data size = 400, Trials = 10
Avg my_bcast time = 0.000011
Avg MPI_Bcast time = 0.000013
```

MPI路障函数

- int MPI_Barrier(MPI_Comm comm)
- 所有进程在执行任务时必须都到达某个同步点才能继续执行后面的内容
- 用来同步程序，使得分布式代码中的某一部分可以精确计时
- 任一个集体通信都是同步的。如果你不能成功地完成MPI_Barrier，那么你也不能成功完成任何集体通信，也会导致程序闲置。

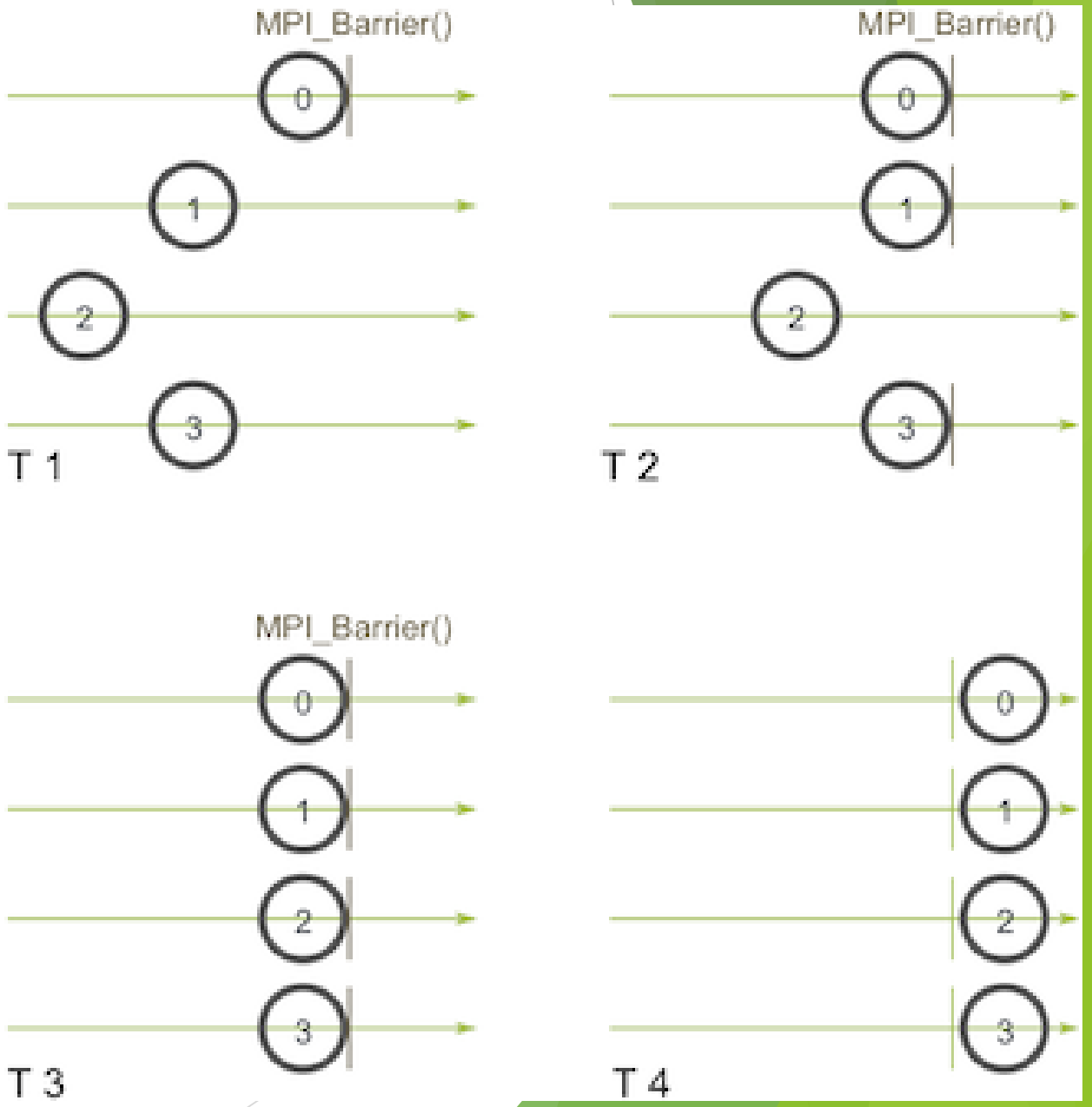


Figure credit: https://mpitutorial.com

MPI分散函数

- MPI分散(scatter)函数将一个进程中的数据按进程的次序(order of process rank)均匀分散给包括自己的所有进程

- 右图为MPI分散函数与MPI广播函数的对比

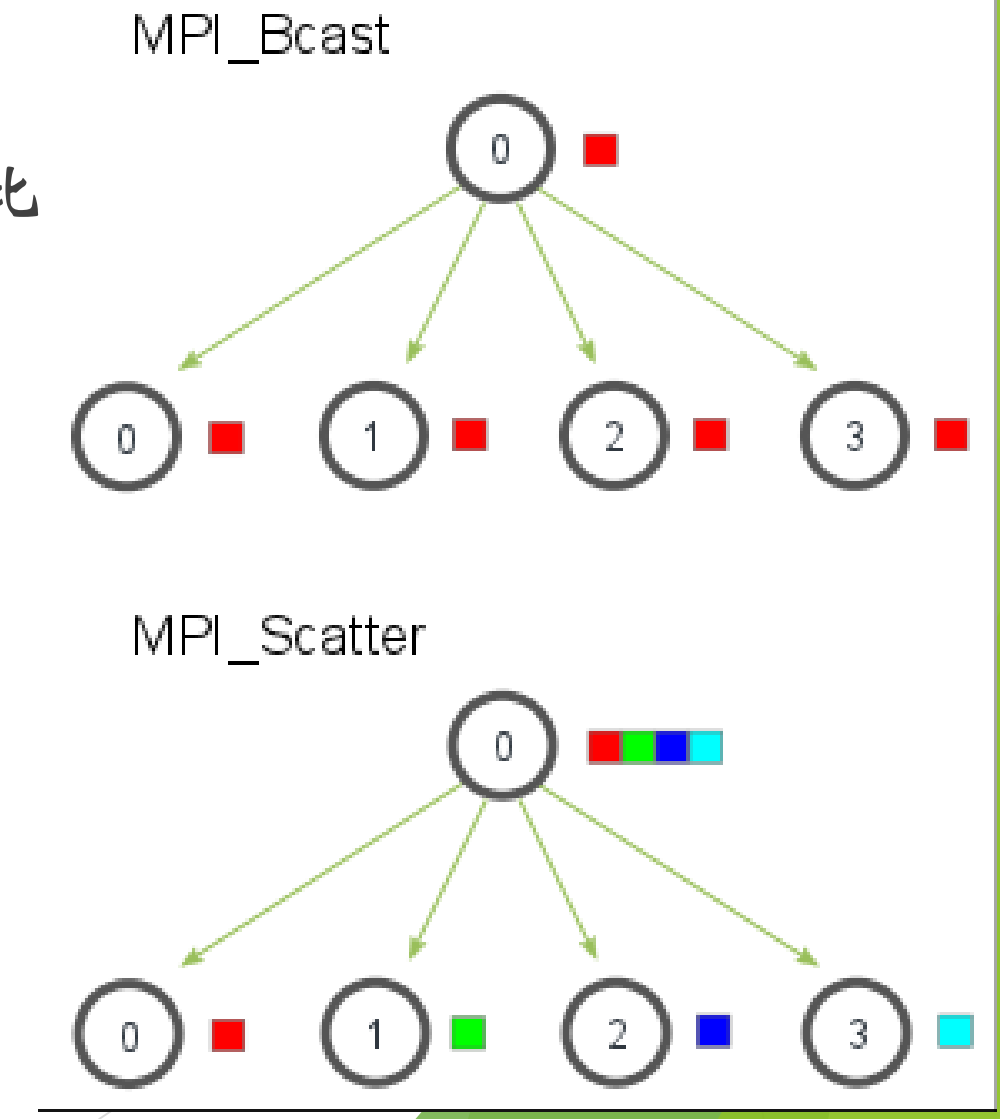


Figure credit: https://mpitutorial.com

MPI分散函数

- int MPI_Scatter(void* sendbuf, int sendcount,
  MPI_Datatype sendtype, void *recvbuff, int recvcount,
  MPI_Datatype recvtype, int root, MPI_Comm comm)
- 所有进程共用此函数
- root表示分散数据的进程rank
- 对于root进程，需要以下参数，
- sendbuf: 发送数据的句柄(handle，句柄是一个对资源的抽象引用，比如指针)
- sendcount: 等分数据后，每块发送的数据量；
  - 数据量为100，分散给5个进程(包括自己),那么sendcount=20
- sendtype:发送数据的类型

MPI分散函数

- int MPI_Scatter(void* sendbuf, int sendcount,
  MPI_Datatype sendtype, void *recvbuff, int recvcount,
  MPI_Datatype recvtype, int root, MPI_Comm comm)
- 对于所有进程，需要以下参数
- recvbuf: 接收数据的句柄
- recvcount:接收数据的个数
- recvtype:接收数据的类型
- 一般有
  sendcout = recvcount=每个进程接收的数据个数，
  sendtype = recvtype

MPI分散函数：例子

- 在进程1中，将向量(1,2,3,...,100)分发给包括自己的四个进程

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


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);

        int *send_data, *recv_data;
        int total_number = 100;
        int recv_num = total_number / world_size;

        if(total_number % world_size != 0 ){
                printf("Data must be equally partitioned!\n");
                MPI_Abort(MPI_COMM_WORLD, 1);
        }

        recv_data = (int *)malloc(sizeof(int)*recv_num);
```

MPI分散函数：例子

```
        if(world_rank == 1){
                send_data = (int *)malloc(sizeof(int)*total_number);
                for(int i=0; i < total_number; i++){
                        send_data[i] = i+1;
                }
        }

        MPI_Scatter(send_data, recv_num, MPI_INT, recv_data, recv_num,
                        MPI_INT, 1, MPI_COMM_WORLD);


        printf("Processor %d received numbers: from %d to %d\n",
                world_rank, recv_data[0], recv_data[recv_num-1]);

        free(recv_data);
        if(world_rank == 1){
                free(send_data);
        }

        MPI_Finalize();
}
```

MPI分散函数：例子

- 输出结果

```
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpicc -o scatter_data sc
atter_data.c
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 3 ./scatter_d
ata
Data must be equally partitioned!Data must be equally partitioned!
application called MPI_Abort(MPI_COMM_WORLD, 1) - process 1
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 4 ./scatter_d
ata
Processor 1 received numbers: from 26 to 50
Processor 2 received numbers: from 51 to 75
Processor 3 received numbers: from 76 to 100
Processor 0 received numbers: from 1 to 25
```

MPI汇总函数

- MPI汇总(gather)函数将多个进程中的数据按进程的次序汇总给其中的某个进程
- 它是MPI分散函数的逆操作

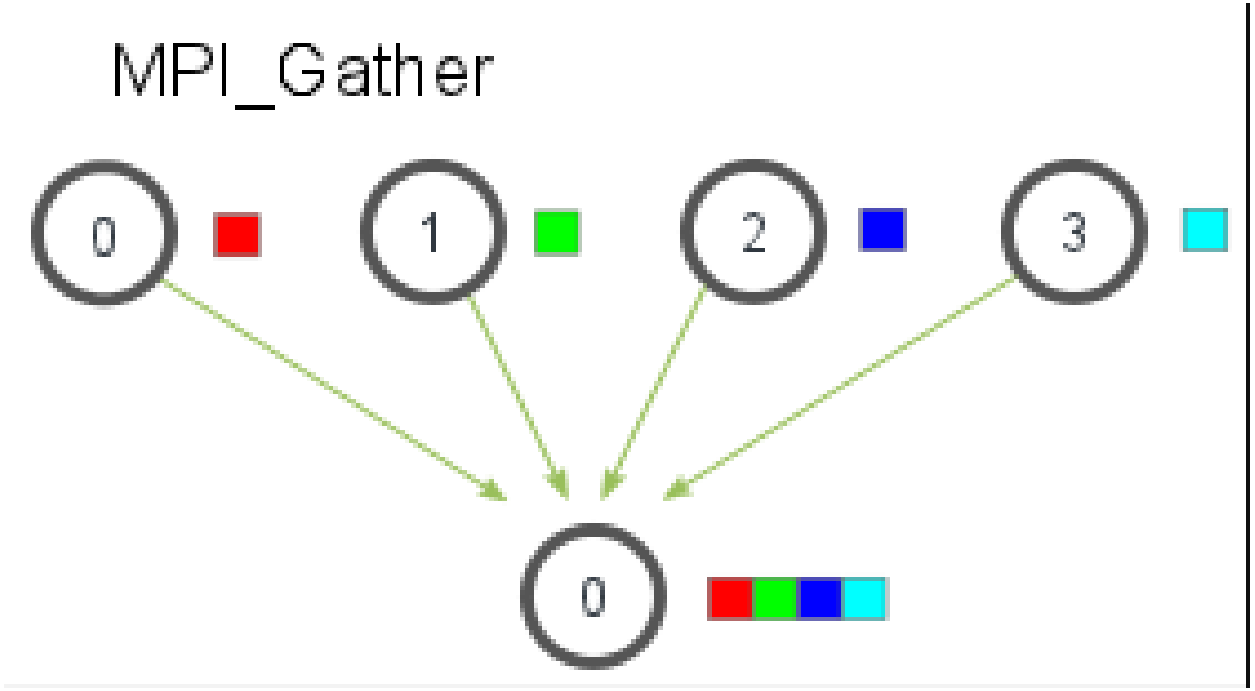


Figure credit: https://mpitutorial.com

MPI汇总函数

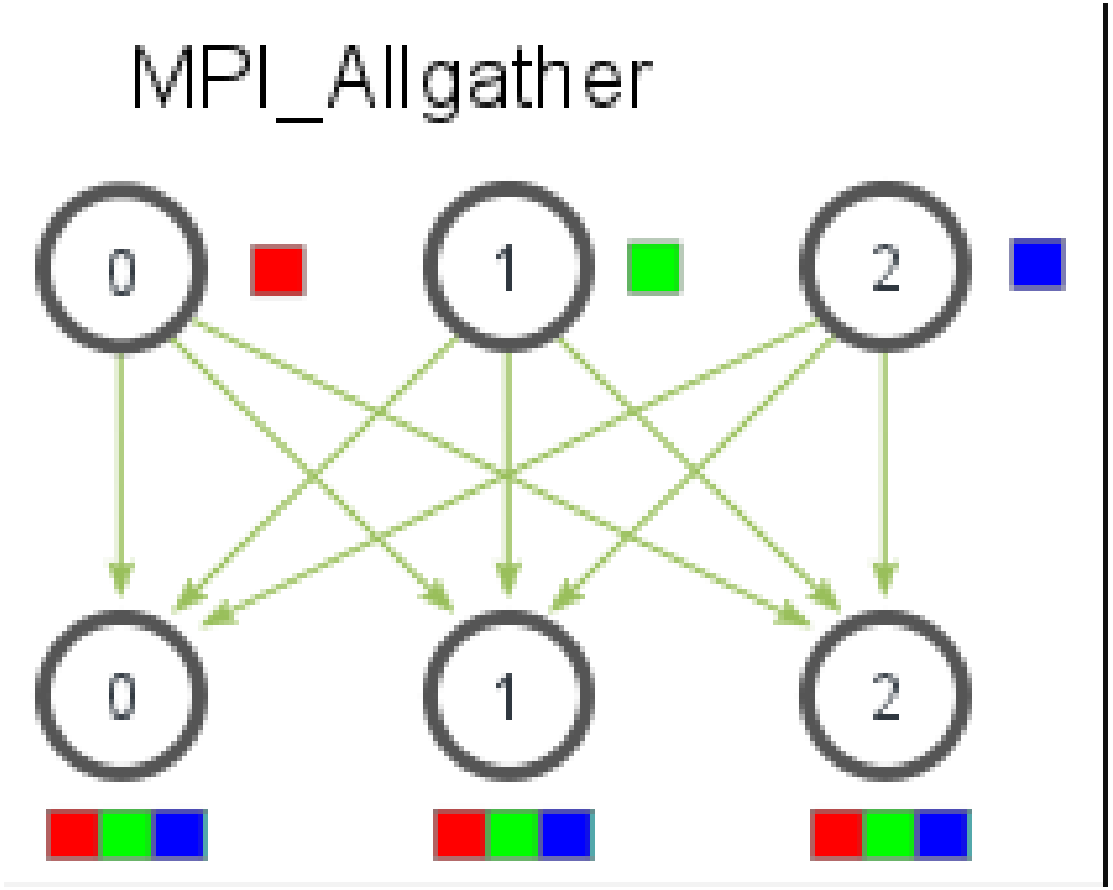
- int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
  void *recvbuff, int recvcount, MPI_Datatype recvtype,
  int root, MPI_Comm comm)；
- 每个参数具有相同的意义，只不过root表示汇总的进程的rank
- 一般地，

  sencount与recvcount相等，均为每个进程发送的数据量

  sendtype与recvtype相等

MPI各个汇总函数

- MPI各个汇总(Allgather)函数将多个进程中的数据汇总,然后发送给所有进程
- MPI_Allgather = MPI_Gather + MPI_Bcast



MPI各个汇总函数

```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                  void *recvbuff, int recvcount, MPI_Datatype recvtype,
                  MPI_Comm comm);
```

不用确定root，因为此执行对于所有进程都是一样的

一般地，

sencount与recvcount相等，均为每个进程发送的数据量

sendtype与recvtype相等

一个例子：Parallel rank problem

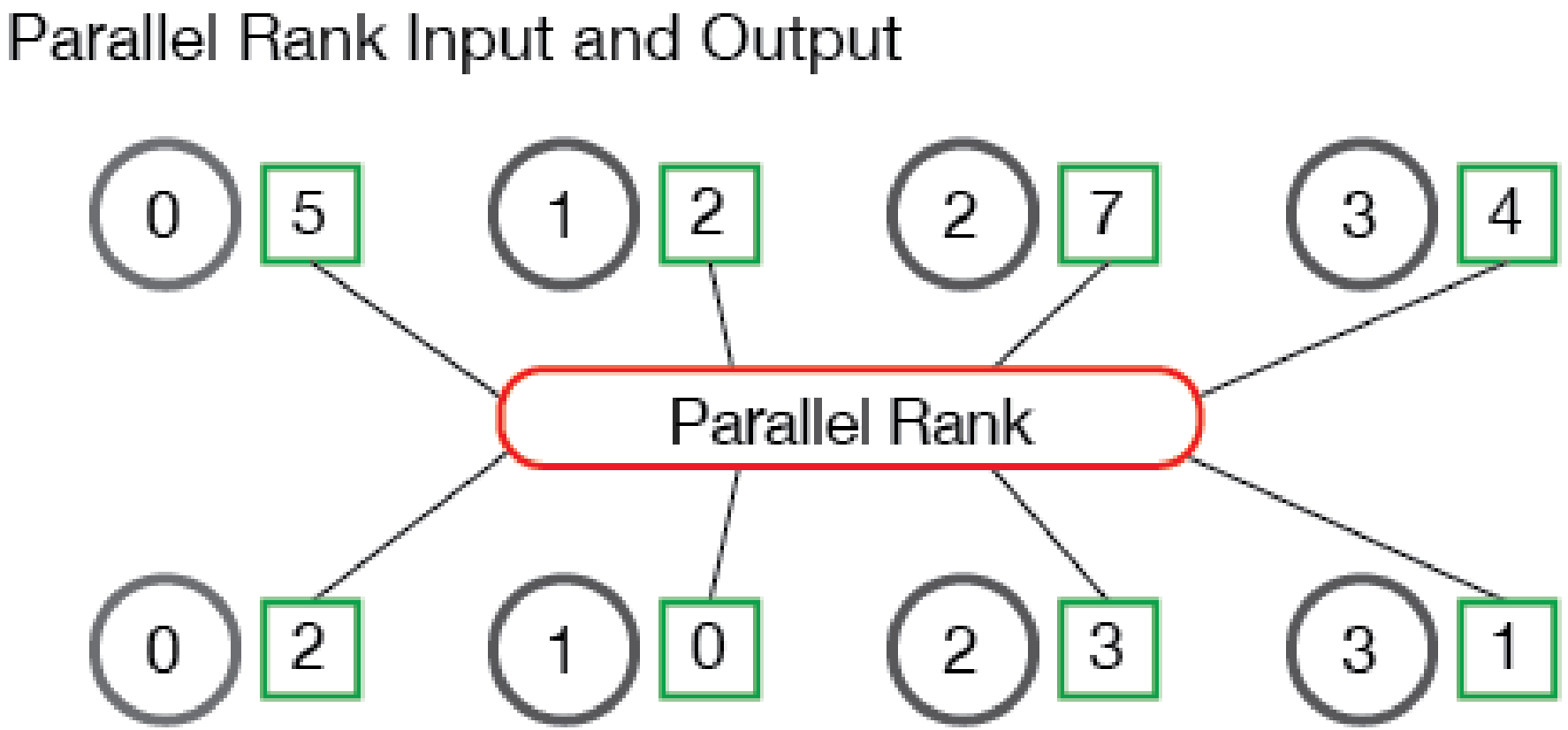


Figure credit: https://mpitutorial.com

一个例子：Parallel rank problem

- 算法设计思路
- Gather到同一个进程
- 找到每个数字的rank
- 将得到的rank结果Scatter到每个进程

- 进程0：5 进程1：2　　进程2：7　　进程3：4
- 进程0：5 2 7 4
- 进程0：2 0 3 1
- 进程0：2 进程1：0　　进程2：3　　进程3：1

一个例子：Parallel rank problem

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

#define MAX_NUMBERS 100

int* order(int* num_array, int length){
        int* tmp = (int *)malloc(sizeof(int)*length);
        for(int i=0; i < length; i++){
                int s = 0;
                for(int j=0; j < length; j++){
                        if(num_array[j] < num_array[i]){
                                s++;
                        }
                }
                tmp[i] = s;
        }
        return tmp;
}

int main(int argc, char** argv){
        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);
```

一个例子：Parallel rank problem

```
int N = 3;
int num_array_per_process[N], num_array[N*world_size];

srand(time(NULL)*world_rank);
for(int i=0; i < N; i++){
        num_array_per_process[i] = (rand()/(float)RAND_MAX)*
                                        MAX_NUMBERS;
}

printf("Process %d has numbers: %d %d %d\n", world_rank,
                num_array_per_process[0], num_array_per_process[1],
                num_array_per_process[2]);

MPI_Gather(num_array_per_process, N, MPI_INT, num_array,
                N, MPI_INT, 0, MPI_COMM_WORLD);


int *num_rank_per_process, *num_rank;
if(world_rank == 0){
        num_rank = order(num_array, N*world_size);
}

num_rank_per_process = (int*) malloc(sizeof(int)*N);

MPI_Scatter(num_rank, N, MPI_INT, num_rank_per_process,
                N, MPI_INT, 0, MPI_COMM_WORLD);
```

一个例子：Parallel rank problem

```
        printf("Process %d: numbers %d %d %d  have global ranks: %d %d %d\n",
                        world_rank,
                        num_array_per_process[0], num_array_per_process[1],
                        num_array_per_process[2],
                        num_rank_per_process[0], num_rank_per_process[1],
                        num_rank_per_process[2]);
        free(num_rank_per_process);
        MPI_Finalize();
}
```

一个例子：Parallel rank problem

- 输出结果

```
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpicc -o parallel_rank p
arallel_rank.c
xiangyu@xiangyu-VirtualBox:~/parallel_computing_files$ mpirun -np 3 ./parallel_
rank
Process 1 has numbers: 96 20 30
Process 2 has numbers: 74 60 2
Process 0 has numbers: 84 39 78
Process 0: numbers 84 39 78  have global ranks: 7 3 6
Process 2: numbers 74 60 2  have global ranks: 5 4 0
Process 1: numbers 96 20 30  have global ranks: 8 1 2
```

MPI缩减操作

- MPI缩减(Reduce)计算操作将若干进程的数字集合通过运算据缩减为其中一个进程上的较小的数字集合
- 比如，四个进程上的数字构成集合{5,2,7,4,}。若用sum函数，得到的结果是18,；若用multiply函数，得到的结果是280

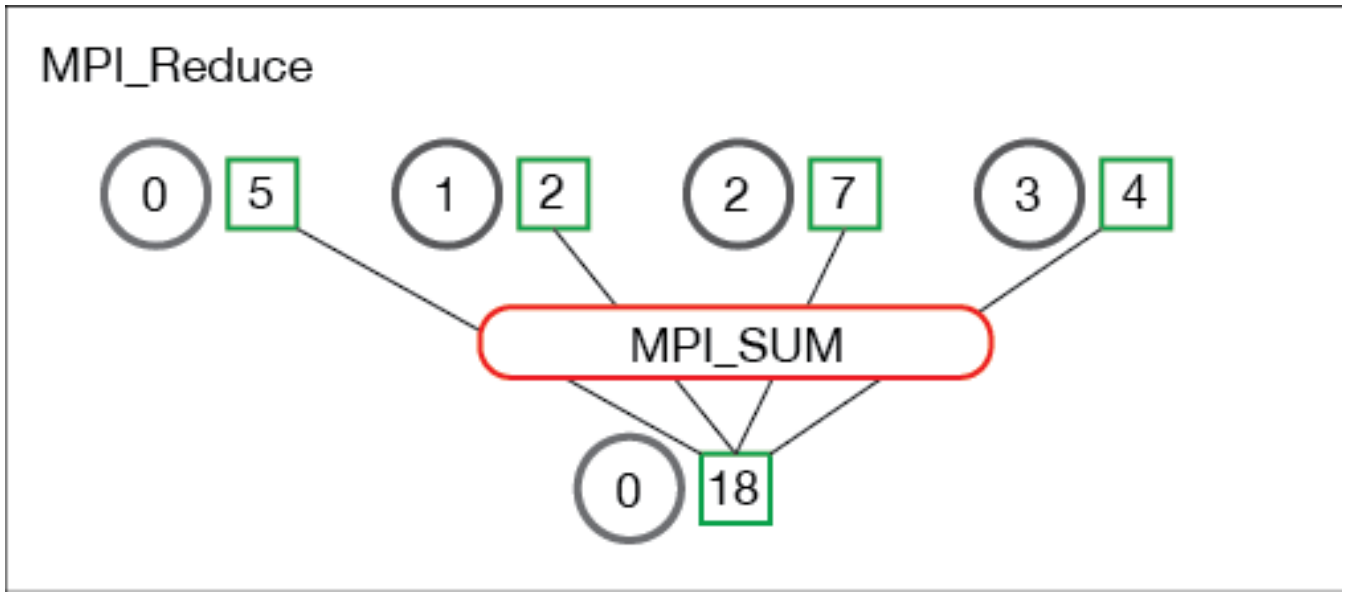


Figure credit: https://mpitutorial.com

MPI缩减操作

- MPI缩减(Reduce)计算操作将若干进程的数字集合通过运算据缩减为其中一个进程上的较小的数字集合
- 对于每个进程具有多个数据

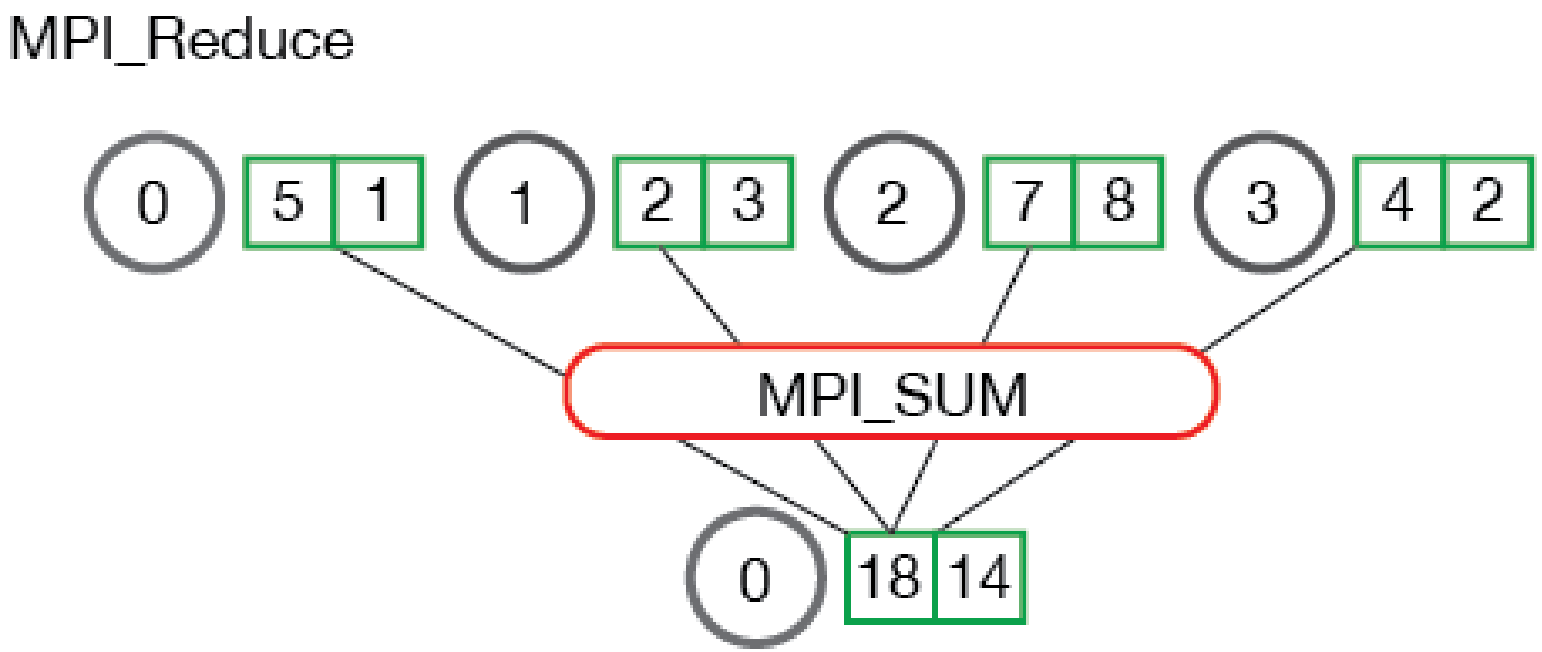


Figure credit: https://mpitutorial.com

MPI缩减操作

- int MPI_Reduce(void *sendbuf, void *recvbuf,
  int count, MPI_Datatype datatype,
  MPI_Op op, int root, MPI_Comm comm)
- sendbuf：发送数据的指针
- recvbuf：进程root中接收数据的指针
- count：发送数据的个数
- datatype：数据类型，且符合缩减操作的要求
- op：将要执行的缩减运算
- root：在指定通信器中接收缩减结果的进程编号
- comm：通信器

MPI缩减操作

- `MPI_MAX` - Returns the maximum element.
- `MPI_MIN` - Returns the minimum element.
- `MPI_SUM` - Sums the elements.
- `MPI_PROD` - Multiplies all elements.
- `MPI_LAND` - Performs a logical *and* across the elements.
- `MPI_LOR` - Performs a logical *or* across the elements.
- `MPI_BAND` - Performs a bitwise *and* across the bits of the elements.
- `MPI_BOR` - Performs a bitwise *or* across the bits of the elements.
- `MPI_MAXLOC` - Returns the maximum value and the rank of the process that owns it.
- `MPI_MINLOC` - Returns the minimum value and the rank of the process that owns it.

Figure credit: https://mpitutorial.com

MPI缩减操作：例子

- 进程0: (8, 5, 6) 进程1: (4, 9, 2) 进程2: (1, 5, 3)
- 找出每个位置在所以进程中的最大值及其所处的进程
- 思路：我们需要定义一个结构体，用于存储值(value)和所处进程编号(rank)
- 比如进程0中， 8 => (8, 0)
- 进程1中， 9 => (9, 1)
- 这样的结构体的数据类型为MPI_2INT
- 若为(double, int)，其数据类型为MPI_DOUBLE_INT