# 并行与分布式计算作业

## 第 6 次作业

姓名：郝裕玮
班级：计科 1 班
学号：18329015

## 对于问题 1：

### 一、问题描述

找出给出的 7 个 MPI 程序中的 bug，并将程序修改正确，比较修改前后的程序运行结果。 MPI 程序见附件。

### 二、解决方案&运行结果

（1）对于 mpi_bug1.c：

错误原因：0 号进程和 1 号进程互发消息时，Send 和 Recv 函数的 tag 不匹配。

该程序错误部分如下：

① 29-31 行：

```
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  printf("Sent to task %d...\n",dest);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD,
&Stat);
```

应修改为：

```
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, 0, MPI_COMM_WORLD);//修改处 1
  printf("Sent to task %d...\n",dest);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, 1, MPI_COMM_WORLD,
&Stat);//修改处 2
```

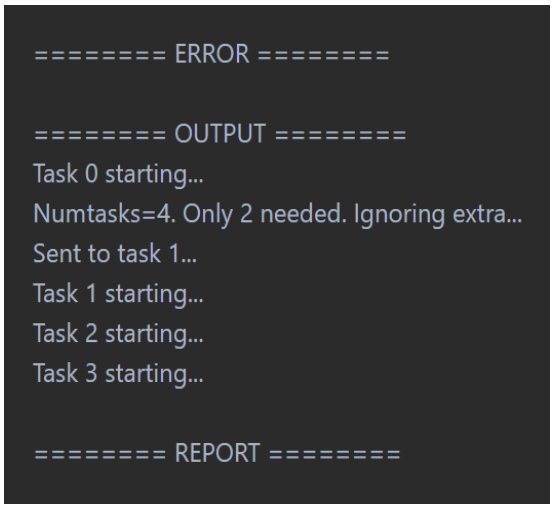
② 39-41 行：

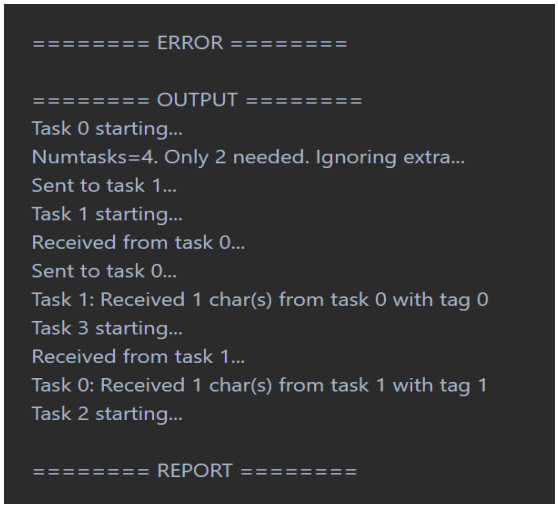
```
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD,
&Stat);
  printf("Received from task %d...\n",source);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
```

应修改为：

```
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, 0, MPI_COMM_WORLD,
&Stat);//修改处 3
  printf("Received from task %d...\n",source);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, 1, MPI_COMM_WORLD);//修改处 4
```

超算习堂上修改前后运行结果如下所示（运行环境：4 核）：




修改后 0 号进程和 1 号进程均收到了对方的消息并进行了回复。

（2）对于 mpi_bug2.c：

错误原因：发送方和接收方的数据类型不匹配（发送方是 int，接收方是 float）。

该程序错误部分如下：

① 16 行：

```
float beta;
```

应修改为：

```
int beta;//修改处 1
```

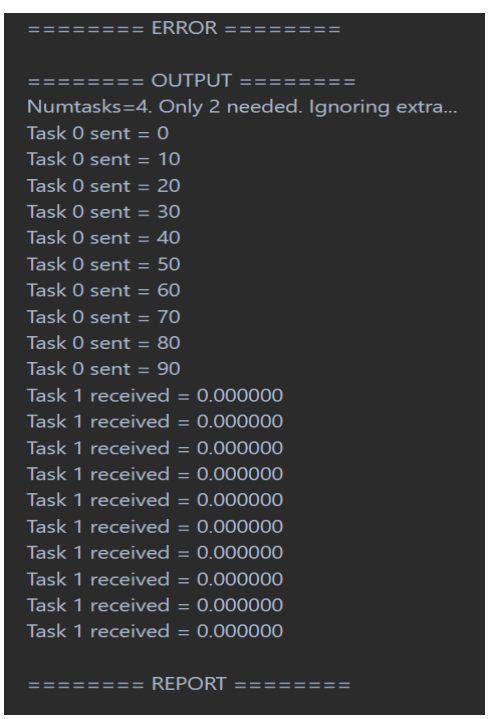
② 37-39 行：

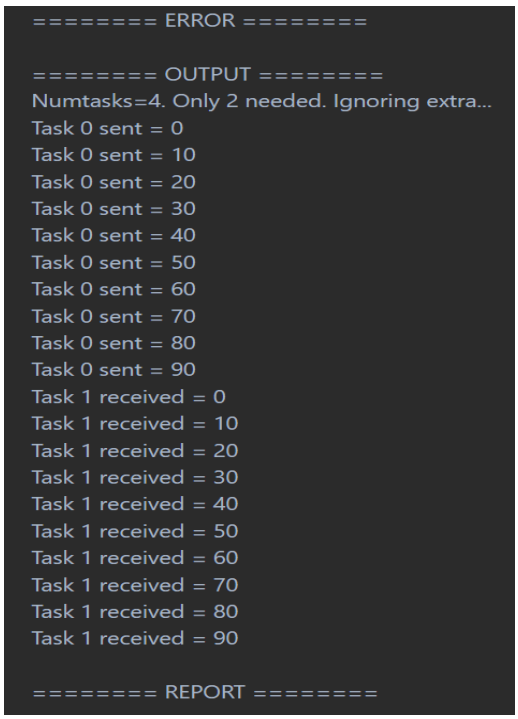
```
    MPI_Irecv(&beta, 1, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &reqs[i]);
    MPI_Wait(&reqs[i], &stats[i]);
    printf("Task %d received = %f\n",rank,beta);
```

应修改为：

```
    MPI_Irecv(&beta, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &reqs[i]);//修
    改处 2
    MPI_Wait(&reqs[i], &stats[i]);
    printf("Task %d received = %d\n",rank,beta);//修改处 3
```

超算习堂上修改前后运行结果如下所示（运行环境：4 核）：




修改后 1 号进程收到了正确的数据。

（3）对于 mpi_bug3.c：

错误原因：MPI 环境未进行初始化和终止。需要添加 MPI_Init 和 MPI_Finalize。

该程序错误部分如下：

① 24-25 行：

```
/***** Initializations *****/
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
```

应修改为：

```
/***** Initializations *****/
MPI_Init(&argc, &argv);//修改处 1
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
```

② 106-108 行：

```
}   /* end of main */
```

应修改为：

```
MPI_Finalize();//修改处 2

}   /* end of main */
```

超算习堂上修改前后运行结果如下所示（运行环境：4 核）：

```
======== ERROR ========
Attempting to use an MPI routine before initializing MPICH
Attempting to use an MPI routine before initializing MPICH
Attempting to use an MPI routine before initializing MPICH
Attempting to use an MPI routine before initializing MPICH

======== OUTPUT ========

======== REPORT ========
```

```
======== ERROR ========

======== OUTPUT ========
MPI task 2 has started...
MPI task 3 has started...
MPI task 0 has started...
MPI task 1 has started...
Initialized array sum = 1.335708e+14
Sent 4000000 elements to task 1 offset= 4000000
Sent 4000000 elements to task 2 offset= 8000000
Task 1 mysum = 4.884048e+13
Sent 4000000 elements to task 3 offset= 12000000
Task 2 mysum = 7.983003e+13
Task 0 mysum = 1.598859e+13
Task 3 mysum = 1.161867e+14
Sample results:
0.000000e+00 2.000000e+00 4.000000e+00 6.000000e+00 8.000000e+00
8.000000e+06 8.000002e+06 8.000004e+06 8.000006e+06 8.000008e+06
1.600000e+07 1.600000e+07 1.600000e+07 1.600001e+07 1.600001e+07
2.400000e+07 2.400000e+07 2.400000e+07 2.400001e+07 2.400001e+07
*** Final sum= 2.608458e+14 ***

======== REPORT ========
```

修改后程序正常运行。

（4）对于 mpi_bug4.c：

错误原因：也应该在 Master 任务中也应当调用 MPI_Reduce，否则 Master 任务中的总和不会被累积。

该程序错误部分如下：

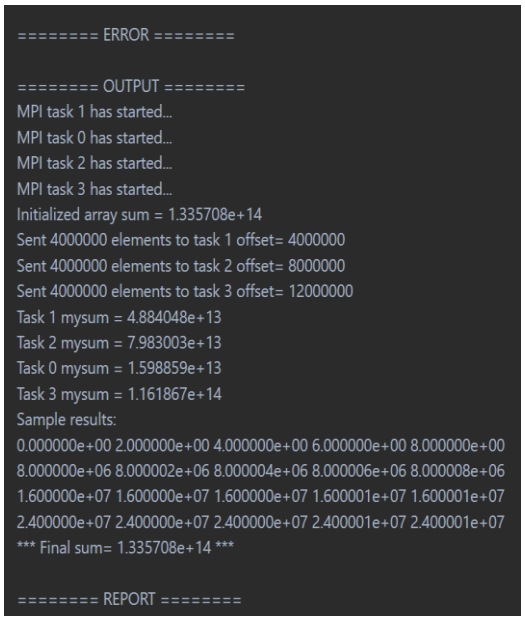
① 69-71 行：

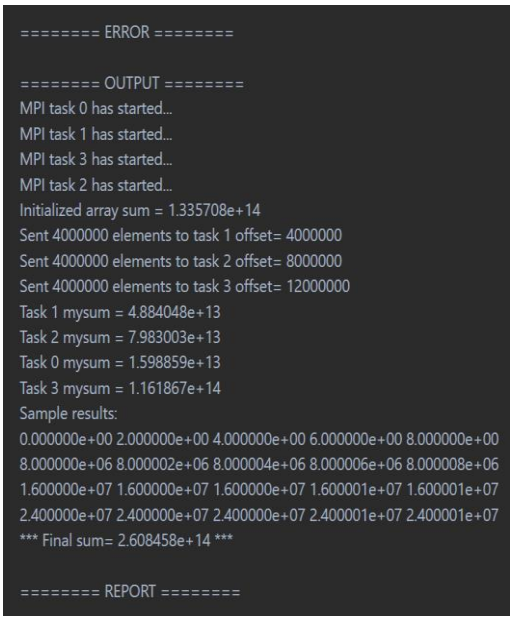
```
    }

  /* Get final sum and print sample results */
```

应修改为：

```
    }
  MPI_Reduce(&mysum, &sum, 1, MPI_FLOAT, MPI_SUM, MASTER,
MPI_COMM_WORLD);//修改处 1

  /* Get final sum and print sample results */
```

超算习堂上修改前后运行结果如下所示（运行环境：4 核）：




修改前 Final sum=Initialized array sum。

修改后 Final sum 得到了正确的数据。

（5）对于 mpi_bug5.c：

错误原因：接收方的缓冲区是有限的。所以该程序可能会因为接收方的缓冲区耗尽导致程序崩溃。所以需要添加 MPI_Barrier，设置

路障以保证一个通信子中所有进程的同步。

且当进程有多个（n > 2）时，我们需要使用冗余的 while 循环来触发其他进程中的 Barrier 以使得进程同步。

该程序错误部分如下：

① 50-52 行：

```
    }
  }
}
```

应修改为：

```
    }
  MPI_Barrier(MPI_COMM_WORLD);//修改处 1
  }
}
```

② 56-64 行：

```
if (rank == 1) {
  while (1) {
    MPI_Recv(data, MSGSIZE, MPI_BYTE, source, tag, MPI_COMM_WORLD,
&status);
    /* Do some work  - at least more than the send task */
    result = 0.0;
    for (i=0; i < 1000000; i++)
      result = result + (double)random();
  }
}
```

应修改为：

```
if (rank == 1) {
  while (1) {
    MPI_Recv(data, MSGSIZE, MPI_BYTE, source, tag, MPI_COMM_WORLD,
&status);
    /* Do some work  - at least more than the send task */
    result = 0.0;
    for (i=0; i < 1000000; i++)
      result = result + (double)random();
  MPI_Barrier(MPI_COMM_WORLD);//修改处 2
  }
}
```

```
//修改处 3
if(rank > 1){
  while(1){
    MPI_Barrier(MPI_COMM_WORLD);
    }
  }
```

超算习堂上修改前后运行结果如下所示（运行环境：4 核）（见下页）：

```
======== ERROR ========

======== OUTPUT ========
mpi_bug5 has started...
INFO: Number of tasks= 4. Only using 2 tasks.
Count= 10 Time= 0.000014 sec.
Count= 20 Time= 0.000005 sec.
Count= 30 Time= 0.000005 sec.
Count= 40 Time= 0.000006 sec.
Count= 50 Time= 0.000006 sec.
Count= 60 Time= 0.000005 sec.
Count= 70 Time= 0.049389 sec.
Count= 80 Time= 0.123454 sec.
Count= 90 Time= 0.124098 sec.
Count= 100 Time= 0.123429 sec.
Count= 110 Time= 0.125802 sec.
Count= 120 Time= 0.136492 sec.
Count= 130 Time= 0.123857 sec.
Count= 140 Time= 0.127315 sec.
Count= 150 Time= 0.128201 sec.
Count= 160 Time= 0.123476 sec.
Count= 170 Time= 0.106382 sec.
Count= 180 Time= 0.104502 sec.
Count= 190 Time= 0.104814 sec.
Count= 200 Time= 0.106773 sec.
Count= 210 Time= 0.109543 sec.
Count= 220 Time= 0.104469 sec.
Count= 230 Time= 0.109474 sec.
Count= 240 Time= 0.104391 sec.
Count= 250 Time= 0.103784 sec.
Count= 260 Time= 0.103976 sec.
Count= 270 Time= 0.115268 sec.
Count= 280 Time= 0.094581 sec.
Count= 290 Time= 0.108176 sec.
Count= 300 Time= 0.111776 sec.
Count= 310 Time= 0.105659 sec.
Count= 320 Time= 0.104437 sec.
Count= 330 Time= 0.116720 sec.
Count= 340 Time= 0.104560 sec.
Count= 350 Time= 0.104691 sec.
Count= 360 Time= 0.104359 sec.
Count= 370 Time= 0.104421 sec.
Count= 380 Time= 0.104346 sec.
Count= 390 Time= 0.111065 sec.
Count= 400 Time= 0.104356 sec.
Count= 410 Time= 0.104768 sec.
Count= 420 Time= 0.107533 sec.
Count= 430 Time= 0.111414 sec.
Count= 440 Time= 0.104622 sec.
Count= 450 Time= 0.106199 sec.
Count= 460 Time= 0.104391 sec.
Count= 470 Time= 0.104423 sec.
Count= 480 Time= 0.104401 sec.
Count= 490 Time= 0.106511 sec.
Count= 500 Time= 0.104982 sec.
Count= 510 Time= 0.105028 sec.

======== REPORT ========
```

```
======== ERROR ========

======== OUTPUT ========
mpi_bug5 has started...
INFO: Number of tasks= 4. Only using 2 tasks.
Count= 10 Time= 0.108254 sec.
Count= 20 Time= 0.119002 sec.
Count= 30 Time= 0.119085 sec.
Count= 40 Time= 0.119318 sec.
Count= 50 Time= 0.118554 sec.
Count= 60 Time= 0.100701 sec.
Count= 70 Time= 0.114100 sec.
Count= 80 Time= 0.101825 sec.
Count= 90 Time= 0.100684 sec.
Count= 100 Time= 0.100654 sec.
Count= 110 Time= 0.100688 sec.
Count= 120 Time= 0.101377 sec.
Count= 130 Time= 0.101055 sec.
Count= 140 Time= 0.101018 sec.
Count= 150 Time= 0.105968 sec.
Count= 160 Time= 0.114100 sec.
Count= 170 Time= 0.110447 sec.
Count= 180 Time= 0.101692 sec.
Count= 190 Time= 0.100667 sec.
Count= 200 Time= 0.101610 sec.
Count= 210 Time= 0.101106 sec.
Count= 220 Time= 0.101008 sec.
Count= 230 Time= 0.100491 sec.
Count= 240 Time= 0.100908 sec.
Count= 250 Time= 0.100560 sec.
Count= 260 Time= 0.110690 sec.
Count= 270 Time= 0.135062 sec.
Count= 280 Time= 0.100674 sec.
Count= 290 Time= 0.100535 sec.
Count= 300 Time= 0.100530 sec.
Count= 310 Time= 0.100776 sec.
Count= 320 Time= 0.101198 sec.
Count= 330 Time= 0.100979 sec.
Count= 340 Time= 0.100703 sec.
Count= 350 Time= 0.100948 sec.
Count= 360 Time= 0.116772 sec.
Count= 370 Time= 0.119141 sec.
Count= 380 Time= 0.118990 sec.
Count= 390 Time= 0.110479 sec.
Count= 400 Time= 0.100863 sec.
Count= 410 Time= 0.100570 sec.
Count= 420 Time= 0.100943 sec.
Count= 430 Time= 0.100703 sec.
Count= 440 Time= 0.100768 sec.
Count= 450 Time= 0.108248 sec.
Count= 460 Time= 0.112977 sec.
Count= 470 Time= 0.100519 sec.

======== REPORT ========
```

题目中 Hint 提示说：

```
* Hint: If possible, try to run the program on two different machines,
* which are connected through a network. You should see uneven timings;
* try to understand/explain them.
```

但我未能理解对于该程序如何实现多机之间的运行（程序主体仅仅是 Send 和 Recv 函数），所以无法通过运行结果来体现程序运行差异，但修改后的代码逻辑肯定是正确的。

（6）对于 mpi_bug6.c：

错误原因：1.进程 1 的偏移量应该从 0 开始，若它从 REPS 开始索引访问，它将超出 reqs 数组的边界。

2.进程 2 是非阻塞的发送接收，所以它不应该调用 MPI_Waitall。

该程序错误部分如下：

① 57 行：

```
    offset = REPS;
```

应修改为：

```
    offset = 0;//修改处 1
```

② 75-100 行：

```
if (rank > 1) {
  nreqs = REPS;

/* Task 2 does the blocking send operation */
  if (rank == 2) {
    dest = 3;
    for (i=0; i<REPS; i++) {
      MPI_Send(&rank, 1, MPI_INT, dest, tag1, COMM);
      if ((i+1)%DISP == 0)
        printf("Task %d has done %d sends\n", rank, i+1);
      }
    }
```

```
/* Task 3 does the non-blocking receive operation */
  if (rank == 3) {
    src = 2;
    offset = 0;
    for (i=0; i<REPS; i++) {
      MPI_Irecv(&buf, 1, MPI_INT, src, tag1, COMM, &reqs[offset]);
      offset += 1;
      if ((i+1)%DISP == 0)
        printf("Task %d has done %d irecvs\n", rank, i+1);
      }
    }

  }
```

应修改为：

```
if (rank > 1) {

/* Task 2 does the blocking send operation */
  if (rank == 2) {
    nreqs = 0;//修改处 2
    dest = 3;
    for (i=0; i<REPS; i++) {
      MPI_Send(&rank, 1, MPI_INT, dest, tag1, COMM);
      if ((i+1)%DISP == 0)
        printf("Task %d has done %d sends\n", rank, i+1);
      }
    }

/* Task 3 does the non-blocking receive operation */
  if (rank == 3) {
    nreqs = REPS;//修改处 3
    src = 2;
    offset = 0;
    for (i=0; i<REPS; i++) {
      MPI_Irecv(&buf, 1, MPI_INT, src, tag1, COMM, &reqs[offset]);
      offset += 1;
      if ((i+1)%DISP == 0)
        printf("Task %d has done %d irecvs\n", rank, i+1);
      }
    }

  }
```

③ 102-103 行：

```
/* Wait for all non-blocking operations to complete and record time */
MPI_Waitall(nreqs, reqs, stats);
```

应修改为：

```
/* Wait for all non-blocking operations to complete and record time */
//修改处 4
  if(rank != 2){
    MPI_Waitall(nreqs, reqs, stats);
  }
```

超算习堂上运行结果如下所示（运行环境：4 核）（见下页）：

```
======== ERROR ========
Fatal error in PMPI_Waitall: Request pending due to failure, error stack:
PMPI_Waitall(392): MPI_Waitall(count=1000, req_array=0x7ffc0a597310, status_array=0x7ffc0a599250) failed
PMPI_Waitall(369): The supplied request in array element 0 was invalid (kind=0)

======== OUTPUT ========
Starting isend/irecv send/irecv test...
Task 0 starting...
Task 1 starting...
Task 2 starting...
Task 3 starting...
Task 3 has done 100 irecvs
Task 3 has done 200 irecvs
Task 3 has done 300 irecvs
Task 3 has done 400 irecvs
Task 3 has done 500 irecvs
Task 3 has done 600 irecvs
Task 1 has done 100 isends/irecvs
Task 3 has done 700 irecvs
Task 3 has done 800 irecvs
Task 3 has done 900 irecvs
Task 3 has done 1000 irecvs
Task 1 has done 200 isends/irecvs
Task 1 has done 300 isends/irecvs
Task 1 has done 400 isends/irecvs
Task 1 has done 500 isends/irecvs
Task 2 has done 100 sends
Task 0 has done 100 isends/irecvs
Task 0 has done 200 isends/irecvs
Task 2 has done 200 sends
Task 0 has done 300 isends/irecvs
Task 2 has done 300 sends
Task 2 has done 400 sends
Task 0 has done 400 isends/irecvs
Task 2 has done 500 sends
Task 0 has done 500 isends/irecvs
Task 2 has done 600 sends
Task 2 has done 700 sends
Task 2 has done 800 sends
Task 2 has done 900 sends
Task 2 has done 1000 sends
Task 1 has done 600 isends/irecvs
Task 1 has done 700 isends/irecvs

======== REPORT ========
```

```
======== ERROR ========

======== OUTPUT ========
Starting isend/irecv send/irecv test...
Task 0 starting...
Task 1 starting...
Task 2 starting...
Task 3 starting...
Task 0 has done 100 isends/irecvs
Task 0 has done 200 isends/irecvs
Task 3 has done 100 irecvs
Task 3 has done 200 irecvs
Task 3 has done 300 irecvs
Task 3 has done 400 irecvs
Task 3 has done 500 irecvs
Task 3 has done 600 irecvs
Task 3 has done 700 irecvs
Task 0 has done 300 isends/irecvs
Task 3 has done 800 irecvs
Task 3 has done 900 irecvs
Task 3 has done 1000 irecvs
Task 0 has done 400 isends/irecvs
Task 0 has done 500 isends/irecvs
Task 1 has done 100 isends/irecvs
Task 1 has done 200 isends/irecvs
Task 1 has done 300 isends/irecvs
Task 2 has done 100 sends
Task 1 has done 400 isends/irecvs
Task 1 has done 500 isends/irecvs
Task 2 has done 200 sends
Task 2 has done 300 sends
Task 2 has done 400 sends
Task 2 has done 500 sends
Task 2 has done 600 sends
Task 2 has done 700 sends
Task 2 has done 800 sends
Task 2 has done 900 sends
Task 2 has done 1000 sends
Task 0 has done 600 isends/irecvs
Task 0 has done 700 isends/irecvs
Task 0 has done 800 isends/irecvs
Task 0 has done 900 isends/irecvs
Task 0 has done 1000 isends/irecvs
Task 1 has done 600 isends/irecvs
Task 1 has done 700 isends/irecvs
Task 1 has done 800 isends/irecvs
Task 1 has done 900 isends/irecvs
Task 1 has done 1000 isends/irecvs
Task 0 time(wall)= 0.002320 sec
Task 1 time(wall)= 0.002322 sec
Task 2 time(wall)= 0.001138 sec
Task 3 time(wall)= 0.001148 sec

======== REPORT ========
```

修改前程序报错。

修改程序正常运行且得到预期结果。

（7）对于 mpi_bug7.c：

错误原因：缓冲区条目数不等于进程的 rank 值，应将 Bcast 中的第 2 个参数的值从 count 改为 numtasks。

该程序错误部分如下：

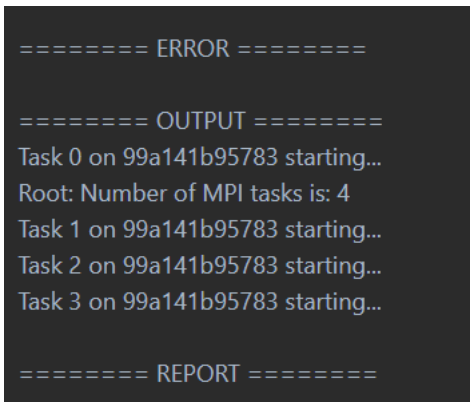
① 29 行：

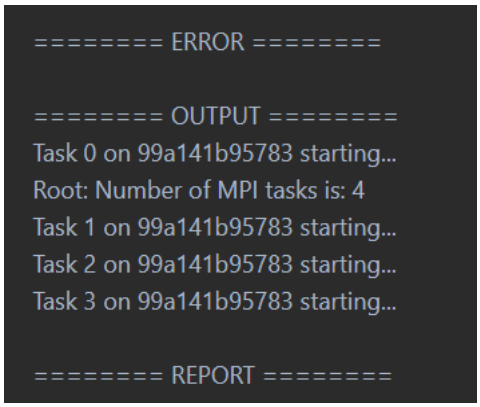
```
MPI_Bcast(&buffer, count, MPI_INT, root, MPI_COMM_WORLD);
```

应修改为：

```
MPI_Bcast(&buffer, numtasks, MPI_INT, root, MPI_COMM_WORLD);//修改处 1
```

超算习堂上运行结果如下所示（运行环境：4 核）：




运行结果无差别，但仍应该对初始代码进行修改。

## 四、遇到的问题及解决方法

该题未遇到难以解决的问题，实验过程较为顺利。

## 对于问题 2：

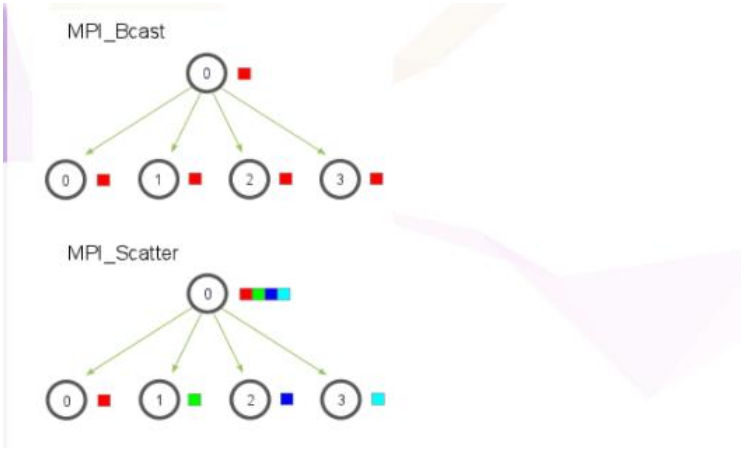
### 一、问题描述

利用 MPI 程序实现 n 个整形数的排序过程，排序算法不限，MPI 的进程不限，可以使用单机运行多个 MPI 进程。提示：在排序的过程中需要 MPI 集合通信如 MPI_Allgather 等。

### 二、解决方案

（1）MPI_Scatter

MPI_Scatter 与 MPI_Bcast 非常相似，都是一对多的通信方式，不同的是后者的 0 号进程将相同的信息发送给所有的进程，而前者则是将一段 array 的不同部分发送给所有的进程，其区别可以用下图概括：




0 号进程分发数据的时候是根据进程的编号进行的，array 中的第一个元素发送给 0 号进程，第二个元素则发送给 1 号进程，以此类推。

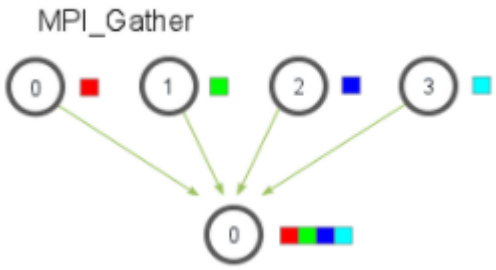
```
MPI_Scatter(
    void* send_data,//存储在 0 号进程的数据，array
    int send_count,//具体需要给每个进程发送的数据的个数
    //如果 send_count 为 1，那么每个进程接收 1 个数据；如果为 2，那么每个进程接收 2 个数据
```

```
    MPI_Datatype send_datatype,//发送数据的类型
    void* recv_data,//接收缓存，缓存 recv_count 个数据
    int recv_count,
    MPI_Datatype recv_datatype,
    int root,//root 进程的编号
    MPI_Comm communicator)
```

通常 send_count 等于 array 的元素个数除以进程个数。

（2）MPI_Gather

MPI_Gather 和 MPI_scatter 刚好相反，他的作用是从所有的进程中将每个进程的数据集中到根进程中，同样根据进程的编号对 array 元素排序，如图所示：




```
MPI_Gather(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,//注意该参数表示的是从单个进程接收的数据个数，不是总数
    MPI_Datatype recv_datatype,
    int root,
    MPI_Comm communicator)
```

代码如下所示（具体思路和详细分析均已包含在代码注释中）：

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

```c
//Merge 函数和 Merge_Sort 函数用于归并排序

/*
  函数功能：合并
  函数参数说明：
  arr: 目标数组
  start: 待合并段的开始下标
  mid: 待合并段的中间下标
  end: 待合并段的结束下标
*/
void Merge(int* arr, int start, int mid, int end)
{
    int len_l, len_r; //左右待合并区间的长度
    len_l = mid - start + 1;
    len_r = end - mid;
    int l[len_l], r[len_r]; //两个临时数组，分别保存待合并的两个区间
    memcpy(l, arr + start, sizeof(int) * len_l);
    memcpy(r, arr + mid + 1, sizeof(int) * len_r);

    int i = 0, j = 0, k = start;
    while(i < len_l && j < len_r)
    {
        arr[k++] = l[i] < r[j] ? l[i++] : r[j++];
    }

    while(i < len_l)
    {
        arr[k++] = l[i++];
    }
}


/*
  函数功能：归并排序
  函数参数说明：
  arr: 待排序的数组
  start: 待排序数组的开始下标
  end: 待排序数组的结束下标
*/
void Merge_Sort(int* arr, int start, int end)
{
    if(start < end)
    {
        int mid = (start + end) / 2;
        //归
```

```
        Merge_Sort(arr, start, mid);
        Merge_Sort(arr, mid + 1, end);
        //并
        Merge(arr, start, mid, end);
    }
}


int main(int argc,char *argv[])
{
    int comm_sz;
    int my_rank;
    int length = 20; //数组大小，可修改
    int size; //每个进程所需数组的大小
    int* sub; //子数组，用于各进程内部的归并排序
    int* result = NULL; //结果数组，存储最终结果
    int i;
    int arr[length]; //待排序的数组

    //MPI 初始化
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    //在分发(Scatter)数据前，若进程号为 0(根进程)，则对数组进行随机数初始化并打印初始数组信息
    if(my_rank == 0){
        printf("初始数组为:\n");
        for(i = 0;i <= length-1; i++){
            arr[i] = rand()%50; //每个元素的值控制在 0-49
            printf("%d ",arr[i]);
        }
        printf("\n");
    }

    size = length / comm_sz; //计算每个进程所需数组的大小
    sub = (int*)malloc(size * sizeof(int)); //为进程所用数组开辟空间

    MPI_Scatter(arr, size, MPI_INT, sub, size, MPI_INT, 0, MPI_COMM_WORLD); //将数组元素分发给每个进程

    Merge_Sort(sub, 0, size-1); //将每个进程分到的子数组进行归并排序
```

```
    //在分发(Scatter)数据后，若进程号为0(根进程)，则开辟结果所需的数组空间（与输入数组一样大）
    if(my_rank == 0) {
        result = (int *)malloc(length * sizeof(int));
    }

    MPI_Gather(sub, size, MPI_INT, result, size, MPI_INT, 0, MPI_COMM_WORLD); //将各进程排序的数组收集到0号进程(根进程)

    //在收集(Gather)各进程数据到根进程后，若进程号为0(根进程)，将收集的结果进行最后一次归并排序
    if (my_rank == 0) {
        Merge_Sort(result, 0, length-1);
        //打印排序后的结果
        printf("\n排序后的结果为:\n");
        for (i = 0;i <= length-1; i++) {
            printf("%d ", result[i]);
        }
        printf("\n");
        free(result);//释放内存
    }
    free(sub);//释放内存

    MPI_Finalize();//终止MPI模拟环境
    return 0;
}
```

## 三、实验结果

超算习堂上运行结果如下所示（运行环境：4核）：

```
======== ERROR ========

======== OUTPUT ========
初始数组为:
0 16 45 5 38 0 4 0 28 2 9 29 49 21 26 28 20 15 33 34

排序后的结果为:
0 0 0 2 4 5 9 15 16 20 21 26 28 28 29 33 34 38 45 49

======== REPORT ========
```

## 四、遇到的问题及解决方法

本次实验主要困难是对 MPI_Scatter 和 MPI_Gather 的应用不够熟练（之前没有专门练习过）。在查阅网上资料和相关例程后完成了对 n 个整数的 MPI 并行归并排序。