中山大学计算机学院2023 年春季超级计算原理与实践MPI 编程作业本科生实验报告

课程名称：超级计算机原理与操作

教学班级	专业（方向）	学号	姓名
2班	计算机科学与技术	21307174	刘俊杰

一、实验题目

MPI编程

二、实验内容

1. 请使用 MPI 函数库实现简单的乒乓游戏。要求使用两个进程模拟两位球手，并使用MPI_Send 和 MPI_Recv 方法来“推挡”信息。该信息在代码中用变量 ping_pong_count 表示，两个进程会轮流成为发送者和接受者，发送者每次发送信息之后 ping_pong_count都会递增 1。部分代码已经给出，请同学们补充完整并打印出每个回合下发送者和接收者的进程标号以及对应的ping_pong_count 值。示例：

```
Process 0 sent and incremented ping_pong_count(value=1) to process 1
Process 0 received ping_pong_count(value=2) from process 1
```

2. 请使用 MPI 函数库实现数组求和的实例。要求进程间通过树结构通信的方式输出求和结果。部分代码已经给出，请同学们补充完整，需要补充的部分参见注释 PLEASE ADD THE RIGHT CODES。本例程中可以首先考虑简单情况，comm_sz 是 2 的幂；进一步可考虑comm_sz 是任意正整数。

3. 请使用 MPI 函数库实现圆周率?的并行计算。（提示：可考虑?的数值计算公式，转化为无穷级数或者定积分的计算。譬如，Π/4=$\displaystyle \int_{0}^1 1/(1+x^2)dx$

三、代码实现以及结果分析

1.实验一

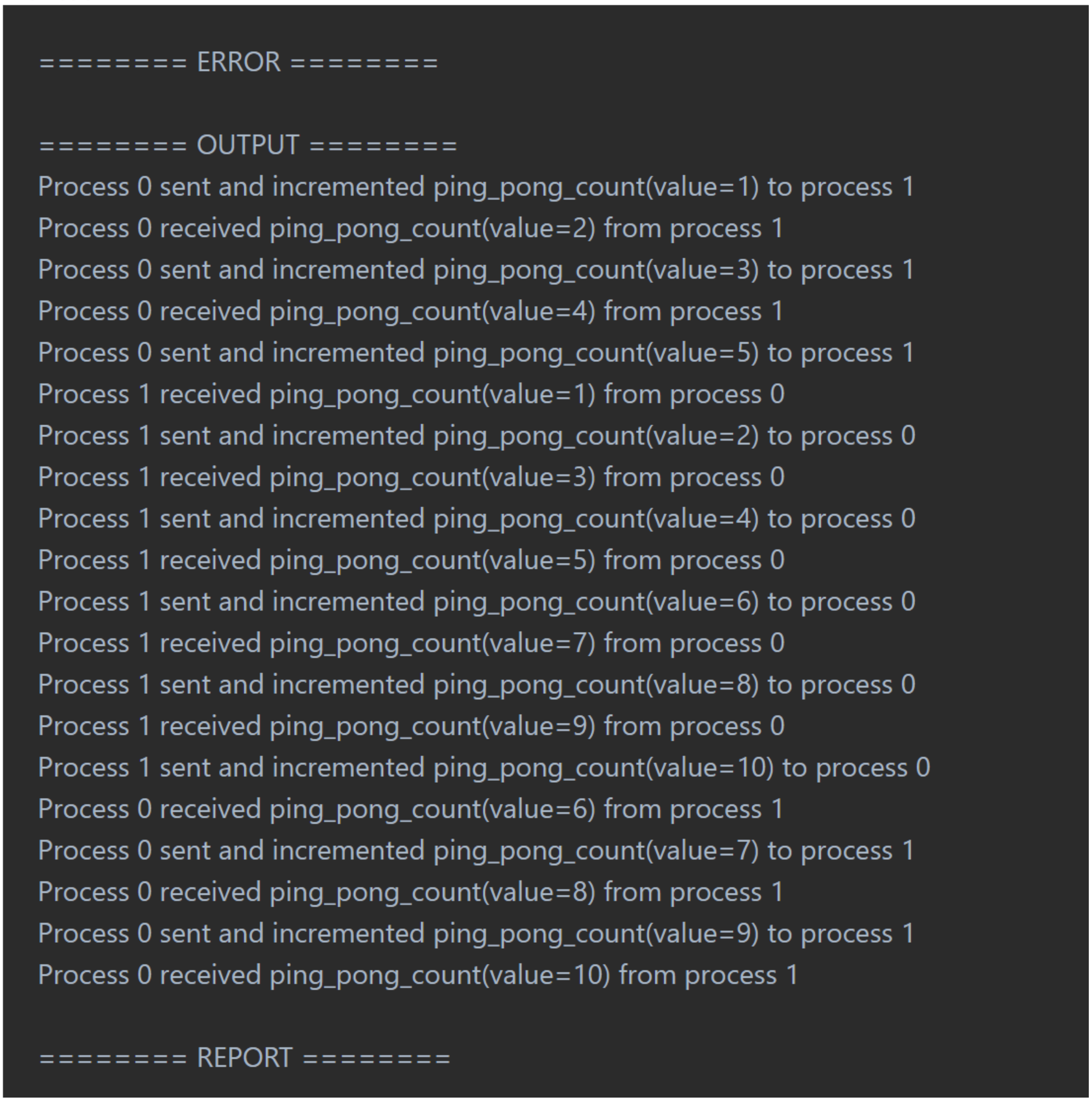
```
/*
 * Purpose:  Using MPI to imitate a table tennis match.
 *
 * Compile:  mpicc -g -Wall -o ex1_ping_pong ex1_ping_pong.c
 * Run:      mpiexec ./ex1_ping_pong
 *
 * Input:    None
 * Output:   None
 *
 * Note:    On each turn, please print out the rank for each process and the value
of "ping_pong_count".
```

```
 */

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

int main(int argc, char** argv) {
  const int PING_PONG_LIMIT = 10;

  // Initialize the MPI environment
  MPI_Init(NULL, NULL);
  // Find out rank, size
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  int world_size;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  // We are assuming 2 processes for this task
  if (world_size != 2) {
    fprintf(stderr, "World size must be two for %s\n", argv[0]);
    MPI_Abort(MPI_COMM_WORLD, 1);
  }

  int ping_pong_count = 0;
  int receive_ping_pong_count=0;
  /********************************************************************/


   while(ping_pong_count<PING_PONG_LIMIT || ping_pong_count==PING_PONG_LIMIT){//大
于limit时退出循环
    if(world_rank==0){
        ping_pong_count++;
        if(ping_pong_count>PING_PONG_LIMIT)break;//大于limit时退出循环
        printf("Process 0 sent and incremented ping_pong_count(value=%d) to
process 1\n",ping_pong_count);
        MPI_Send(&ping_pong_count,1,MPI_INT,1,0,MPI_COMM_WORLD);//向1核发送消息即
ping_pong_count

MPI_Recv(&ping_pong_count,1,MPI_INT,1,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);//接收1
核消息
        if(ping_pong_count>PING_PONG_LIMIT)break;//大于limit时退出循环
        printf("Process 0 received ping_pong_count(value=%d) from process
1\n",ping_pong_count);

    }
    else{

MPI_Recv(&ping_pong_count,1,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);//接收0
核消息
        printf("Process 1 received ping_pong_count(value=%d) from process
0\n",ping_pong_count);
        ping_pong_count++;
        if(ping_pong_count>PING_PONG_LIMIT)break;//大于limit时退出循环
        printf("Process 1 sent and incremented ping_pong_count(value=%d) to
process 0\n",ping_pong_count);
        MPI_Send(&ping_pong_count,1,MPI_INT,0,1,MPI_COMM_WORLD);//向0核发送消息即
ping_pong_count


    }
   }


   /*******************************************************************/
  MPI_Finalize();
  return 0;
}
/*
Process 0 sent and incremented ping_pong_count(value=1) to process 1
Process 0 received ping_pong_count(value=2) from process 1
*/
```

实验原理

0核先打印输出后发送数据给1核，再接收1核的消息，然后ping_pong_count+=1,若未收到则阻塞；1核先接收0核的消息，若未收到则阻塞，收到后输出打印,ping_pong_count+=1，然后再发送，不断循环，直到ping_pong_count>10退出。

运行结果



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

======== OUTPUT ========
Process 0 sent and incremented ping_pong_count(value=1) to process 1
Process 0 received ping_pong_count(value=2) from process 1
Process 0 sent and incremented ping_pong_count(value=3) to process 1
Process 0 received ping_pong_count(value=4) from process 1
Process 0 sent and incremented ping_pong_count(value=5) to process 1
Process 1 received ping_pong_count(value=1) from process 0
Process 1 sent and incremented ping_pong_count(value=2) to process 0
Process 1 received ping_pong_count(value=3) from process 0
Process 1 sent and incremented ping_pong_count(value=4) to process 0
Process 1 received ping_pong_count(value=5) from process 0
Process 1 sent and incremented ping_pong_count(value=6) to process 0
Process 1 received ping_pong_count(value=7) from process 0
Process 1 sent and incremented ping_pong_count(value=8) to process 0
Process 1 received ping_pong_count(value=9) from process 0
Process 1 sent and incremented ping_pong_count(value=10) to process 0
Process 0 received ping_pong_count(value=6) from process 1
Process 0 sent and incremented ping_pong_count(value=7) to process 1
Process 0 received ping_pong_count(value=8) from process 1
Process 0 sent and incremented ping_pong_count(value=9) to process 1
Process 0 received ping_pong_count(value=10) from process 1

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

问题(疑问)：输出顺序不确定且不为期待的顺序

2.实验二

```
/*
 * Purpose:  Use tree-structured communication to find the global sum
 *           of a random collection of ints
 *
 * Compile:  mpicc -g -Wall -o ex2_tree_sum ex2_tree_sum.c
 * Run:      mpiexec -n <comm_sz> ./ex2_tree_sum
 *
 * Input:    None
 * Output:   Random values generated by processes, and their global sum.
 *
 * Note:     This version assumes comm_sz is a power of 2.
 */


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

int Global_sum(int my_int, int my_rank, int comm_sz, MPI_Comm comm);

const int MAX_CONTRIB = 20;

int main(void) {
   int i, sum, my_int;
   int my_rank, comm_sz;
   MPI_Comm comm;
   int* all_ints = NULL;

   MPI_Init(NULL, NULL);
   comm = MPI_COMM_WORLD;
   MPI_Comm_size(comm, &comm_sz);
   MPI_Comm_rank(comm, &my_rank);

   srandom(my_rank + 1);
   my_int = random() % MAX_CONTRIB;

   sum = Global_sum(my_int, my_rank, comm_sz, comm);

   if ( my_rank == 0) {
      all_ints = malloc(comm_sz*sizeof(int));
      MPI_Gather(&my_int, 1, MPI_INT, all_ints, 1, MPI_INT, 0, comm);
      printf("Ints being summed:\n   ");
      for (i = 0; i < comm_sz; i++)
         printf("%d ", all_ints[i]);
      printf("\n");
      printf("Sum = %d\n",sum);
      free(all_ints);
   } else {
      MPI_Gather(&my_int, 1, MPI_INT, all_ints, 1, MPI_INT, 0, comm);
   }

   MPI_Finalize();
   return 0;
}  /* main */

/*-------------------------------------------------------------------
 * Function:   Global_sum
 * Purpose:    Implement a global sum using tree-structured communication
 * Notes:
 * 1.  comm_sz must be a power of 2
 * 2.  The return value is only valid on process 0
 */
int Global_sum(
      int my_int    /* in */,
      int my_rank   /* in */,
      int comm_sz   /* in */,
      MPI_Comm comm /* in */) {
 int send_sum=my_int;//发送数据
 int n = 1;
 while (n < comm_sz) {
   int receive_flag=0;//是否接收过消息
   int send_flag=0;//是否发送过消息
   int receive_or_send_rank = my_rank ^ n;
   if (my_rank < receive_or_send_rank&&receive_flag==0) {
      int received_sum;//接收数据
      MPI_Recv(&received_sum, 1, MPI_INT, receive_or_send_rank, 0, comm,
MPI_STATUS_IGNORE);
      send_sum+= received_sum;
   }
   else if(send_flag==0){
      send_flag=1;
      MPI_Send(&send_sum, 1, MPI_INT, receive_or_send_rank, 0, comm);
 }
 n*=2;
}
return send_sum;
}  /* Global_sum */
```

实验原理

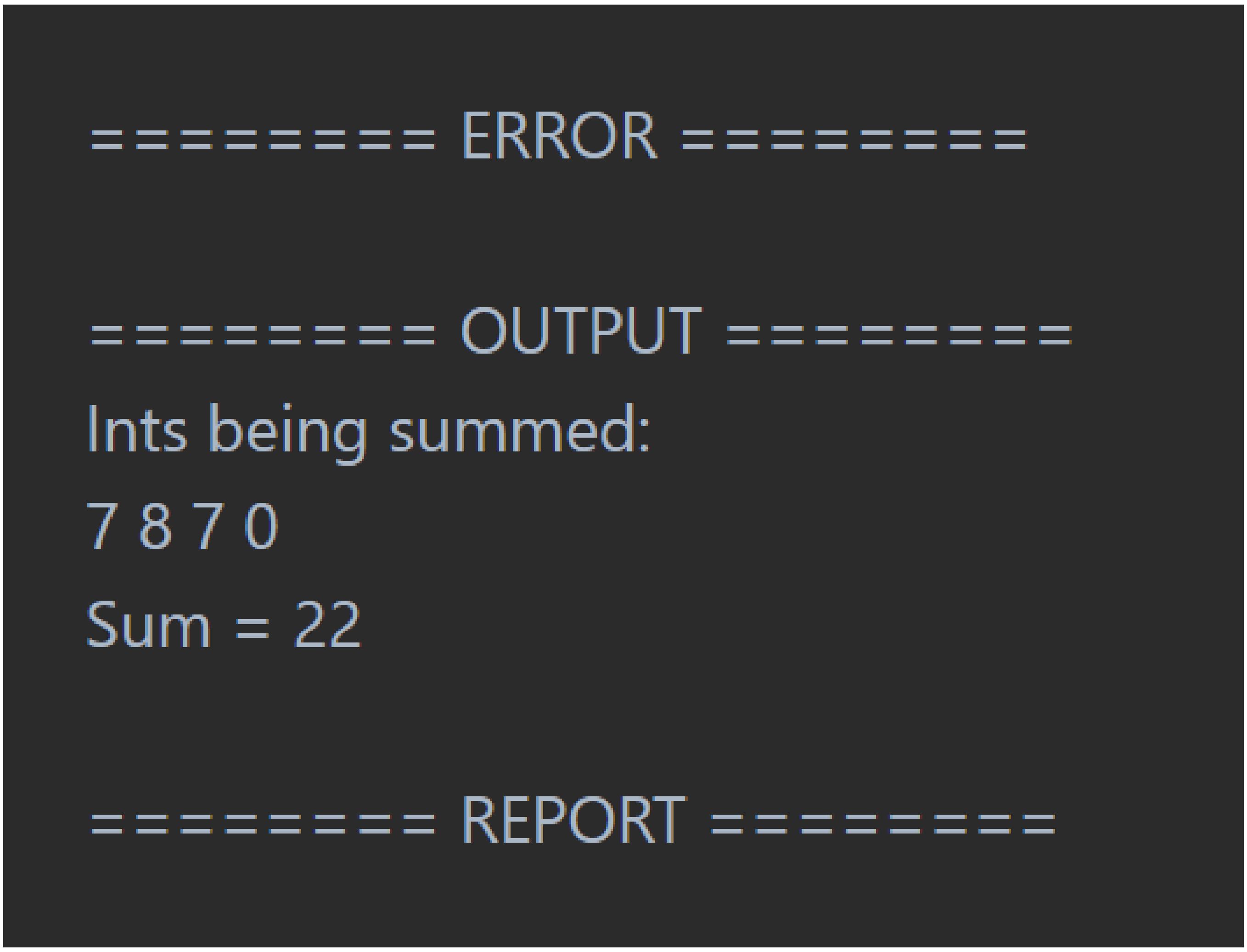
设进程号rank，则通过对课件上树形求和的传递接收消息之间的进程号，我们得到接收和发送消息的进程号如何求，如下面代码所示.

```
int send_sum=my_int;//发送数据
 int n = 1;
 while (n < comm_sz) {
   int receive_flag=0;//是否接收过消息
   int send_flag=0;//是否发送过消息
   int receive_or_send_rank = my_rank ^ n;
   if (my_rank < receive_or_send_rank&&receive_flag==0) {
      int received_sum;//接收数据
      MPI_Recv(&received_sum, 1, MPI_INT, receive_or_send_rank, 0, comm,
MPI_STATUS_IGNORE);
      send_sum+= received_sum;
   }
   else if(send_flag==0){
      send_flag=1;
      MPI_Send(&send_sum, 1, MPI_INT, receive_or_send_rank, 0, comm);
 }
 n*=2;
}
```

得到进程号后我们便可以进行数据的传递与接收。由于我们先得到的是接收消息的进程号，所以会进行消息的接收而阻塞，所以除最后一个进程不会出现还未接收消息相加，就开始传递的现象。

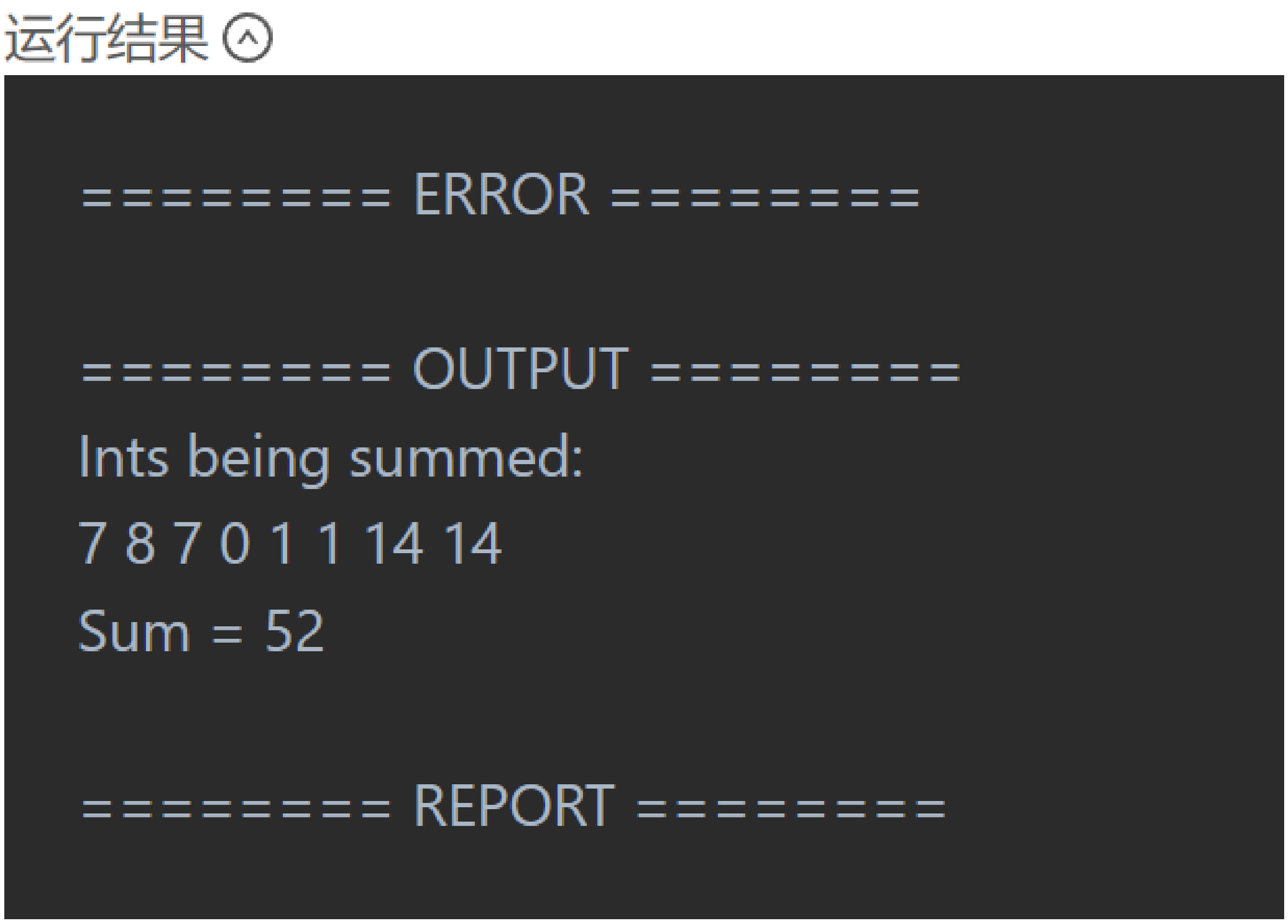
运行结果

用四核运行



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

========= OUTPUT =========
Ints being summed:
7 8 7 0
Sum = 22

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

用八核运行



运行结果

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

========= OUTPUT =========
Ints being summed:
7 8 7 0 1 1 14 14
Sum = 52

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

3.实验三

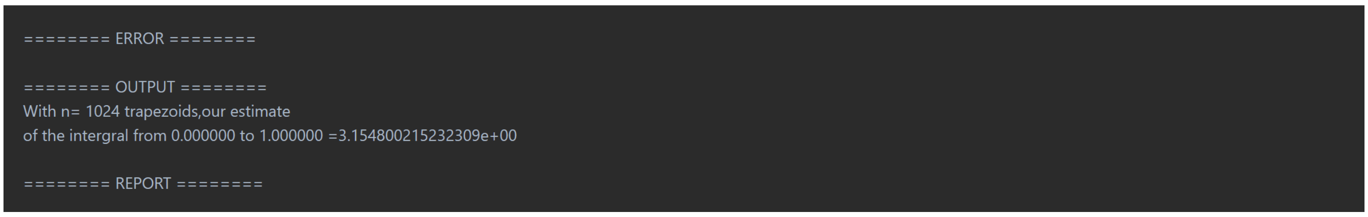
```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
double f(double x){//返回f(x)=1/(1+x^2)
 return 1/(double)(1+x*x);
}
double Trap(double left_endpt,double right_endpt,int trap_count,double base_len){
    double estimate,x;
    int i;
    estimate=(f(left_endpt)+f(right_endpt))/2.0;
    for (i=0;i<=trap_count-1;i++){
        x=left_endpt+i*base_len;
        estimate+=f(x);
    }
    return estimate*base_len;
}
int main(int argc, char** argv) {
int my_rank,comm_sz,n=1024,local_n;//将积分区域分为n个矩形计算
double a=0.0,b=1.0,h,local_a,local_b;//积分区间[a,b]
double local_int,total_int;
int source;
MPI_Init(NULL,NULL);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_Comm_size(MPI_COMM_WORLD,&comm_sz);
h=(b-a)/n;
local_n=n/comm_sz;
local_a=a+my_rank*local_n*h;//当前进程的左区间
local_b=local_a+local_n*h;//当前进程的右区间
local_int=Trap(local_a,local_b,local_n,h);//计算出当前进程需要计算的矩形面积之和
if(my_rank!=0){
    MPI_Send(&local_int,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD);//向0号进程发送当前进程需要计算的矩形面积之和
}
else{
    total_int=local_int;
    for(source=1;source<comm_sz;source++){//接收其他进程的传递结果，并进行相加

MPI_Recv(&local_int,1,MPI_DOUBLE,source,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        total_int+=local_int;
    }
}
if(my_rank==0){//0号进程打印输出估计结果
    printf("With n= %d trapezoids,our estimate\n",n);
    printf("of the intergral from %f to %f =%.15e \n",a,b,total_int*4);
}
MPI_Finalize();
return 0;
}
```

实验原理

首先我们先将积分区域分成多个矩形计算，再将这些矩形平均分给不同的进程计算，最后将计算的总和传递到0号进程中，0号进程接受后打印输出估计。

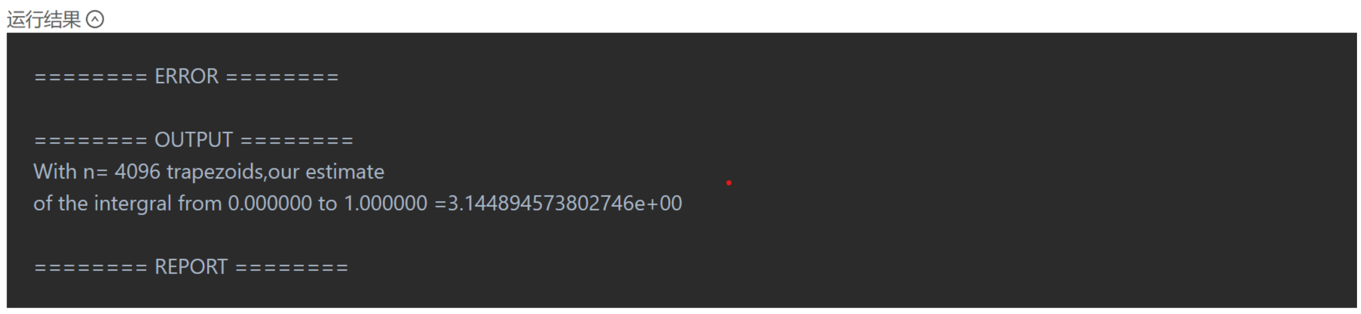
###运行结果

分成1024个矩形估计



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

======== OUTPUT ========
With n= 1024 trapezoids,our estimate
of the intergral from 0.000000 to 1.000000 =3.154800215232309e+00

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

分成4096个矩形估计


运行结果

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

======== OUTPUT ========
With n= 4096 trapezoids,our estimate
of the intergral from 0.000000 to 1.000000 =3.144894573802746e+00

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

可以从运行结果看出分成4096个矩形比1024个矩形的计算估计的误差更加小。

四、学习心得

通过本次实验，我对MPI的信息的传递和接收，以及使用多个进程解决问题的理解更加深入。同时初步学会了使用MPI编程解决一些基本的问题。