HERALD COLLEGE KATHMANDU

ing GROUP

Module Title: Distributed and Cloud Systems Programming

(5CS022)

Subject Title: Workshop 02

Student Name: Nayan Raj Khanal

Student Code: 2227486

Instructor: Mr. Prabin Sapkota

Submitted on: 17.03.2023

University of Wolverhampton
School of Mathematics and Computer Science

5CS022 Distribute and Cloud Systems Programming Week 2 Workshop

Overview

The aim of this workshop is to build on the previous workshop with MPI and to carry out an assessed task. You can carry out this workshop either the university servers: thinlinc.wlv.ac.uk or your own Linux system (if you prefer to use Putty to log in instead of the Thinlinc client, connect to the **server tl-01.wlv.ac.uk** instead).

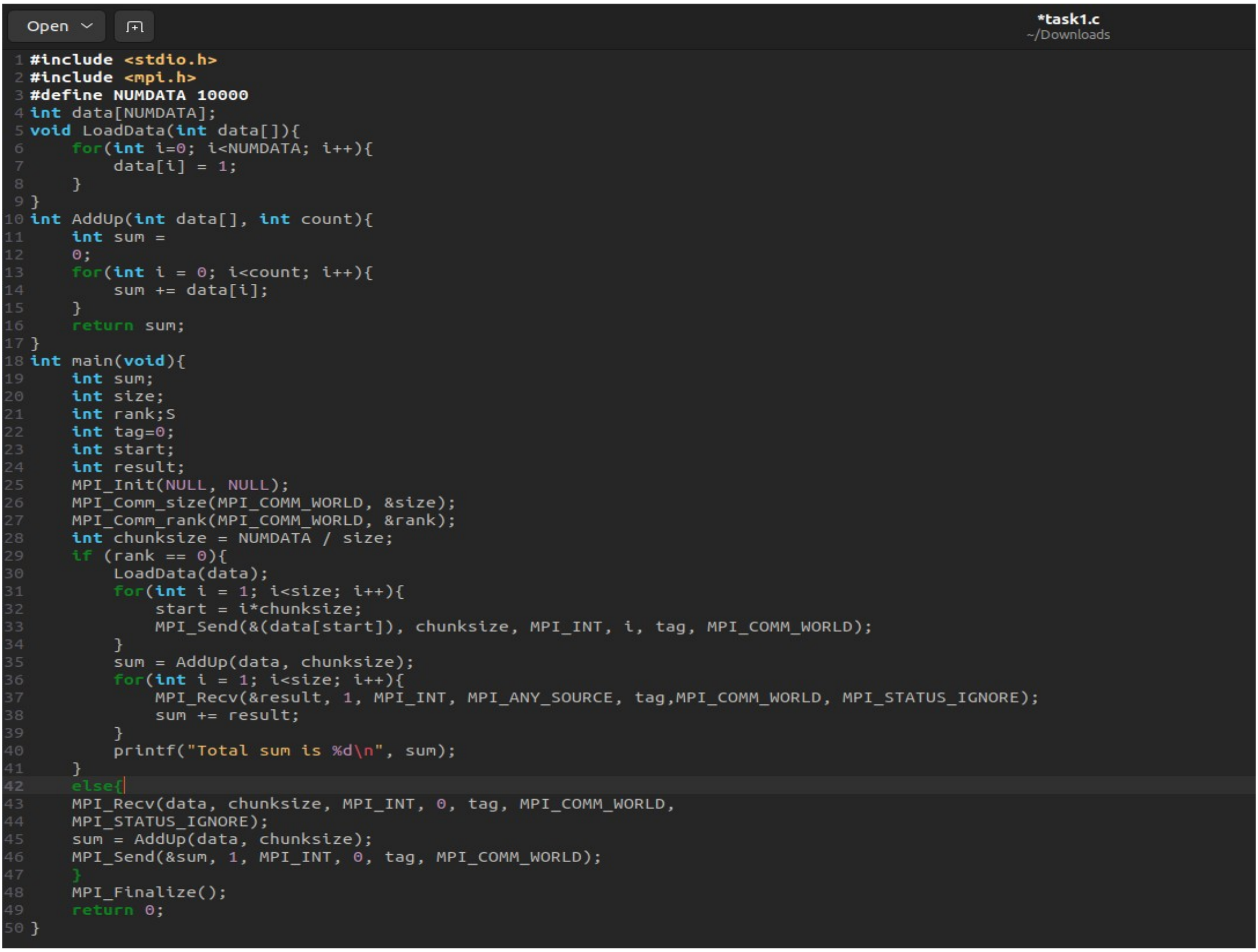
Tasks

1. The following C program sums up all the values in array "data" and displays the sum total.:

```
#include <stdio.h>

#define NUMDATA 10000
int data[NUMDATA];

void LoadData(int data[])
{
  for(int i = 0; i < NUMDATA; i++){
    data[i] = 1;
  }
}

int AddUp(int data[], int count)
{
  int sum = 0;
  for(int i = 0; i < count; i++){
    sum += data[i];
  }
  return sum;
}

int main(void) {
  int sum;

  LoadData(data);
  sum = AddUp(data, NUMDATA);
  printf("The total sum of data is %d\n", sum);
  return 0;
}
```

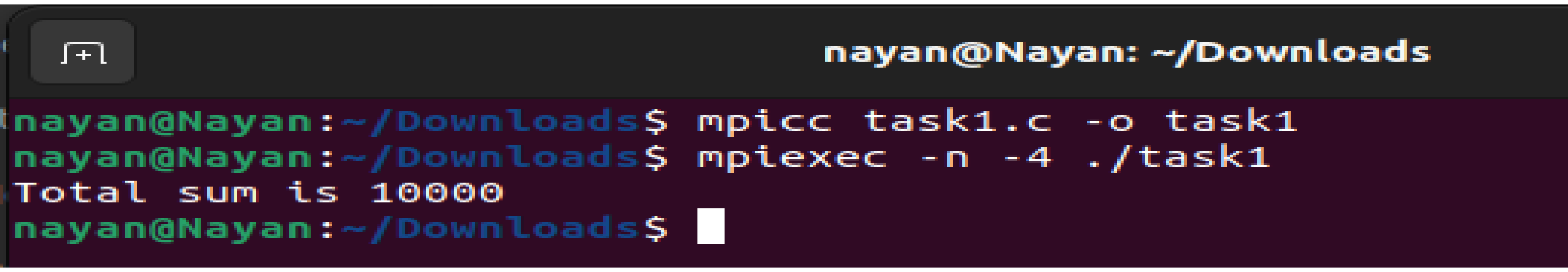
   Convert it to MPI to run with any number of nodes including just one.

Code



```c
#include <stdio.h>
#include <mpi.h>
#define NUMDATA 10000
int data[NUMDATA];
void LoadData(int data[]){
    for(int i=0; i<NUMDATA; i++){
        data[i] = 1;
    }
}
int AddUp(int data[], int count){
    int sum =
    0;
    for(int i = 0; i<count; i++){
        sum += data[i];
    }
    return sum;
}
int main(void){
    int sum;
    int size;
    int rank;S
    int tag=0;
    int start;
    int result;
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int chunksize = NUMDATA / size;
    if (rank == 0){
        LoadData(data);
        for(int i = 1; i<size; i++){
            start = i*chunksize;
            MPI_Send(&(data[start]), chunksize, MPI_INT, i, tag, MPI_COMM_WORLD);
        }
        sum = AddUp(data, chunksize);
        for(int i = 1; i<size; i++){
            MPI_Recv(&result, 1, MPI_INT, MPI_ANY_SOURCE, tag,MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            sum += result;
        }
        printf("Total sum is %d\n", sum);
    }
    else{
    MPI_Recv(data, chunksize, MPI_INT, 0, tag, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
    sum = AddUp(data, chunksize);
    MPI_Send(&sum, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

Output



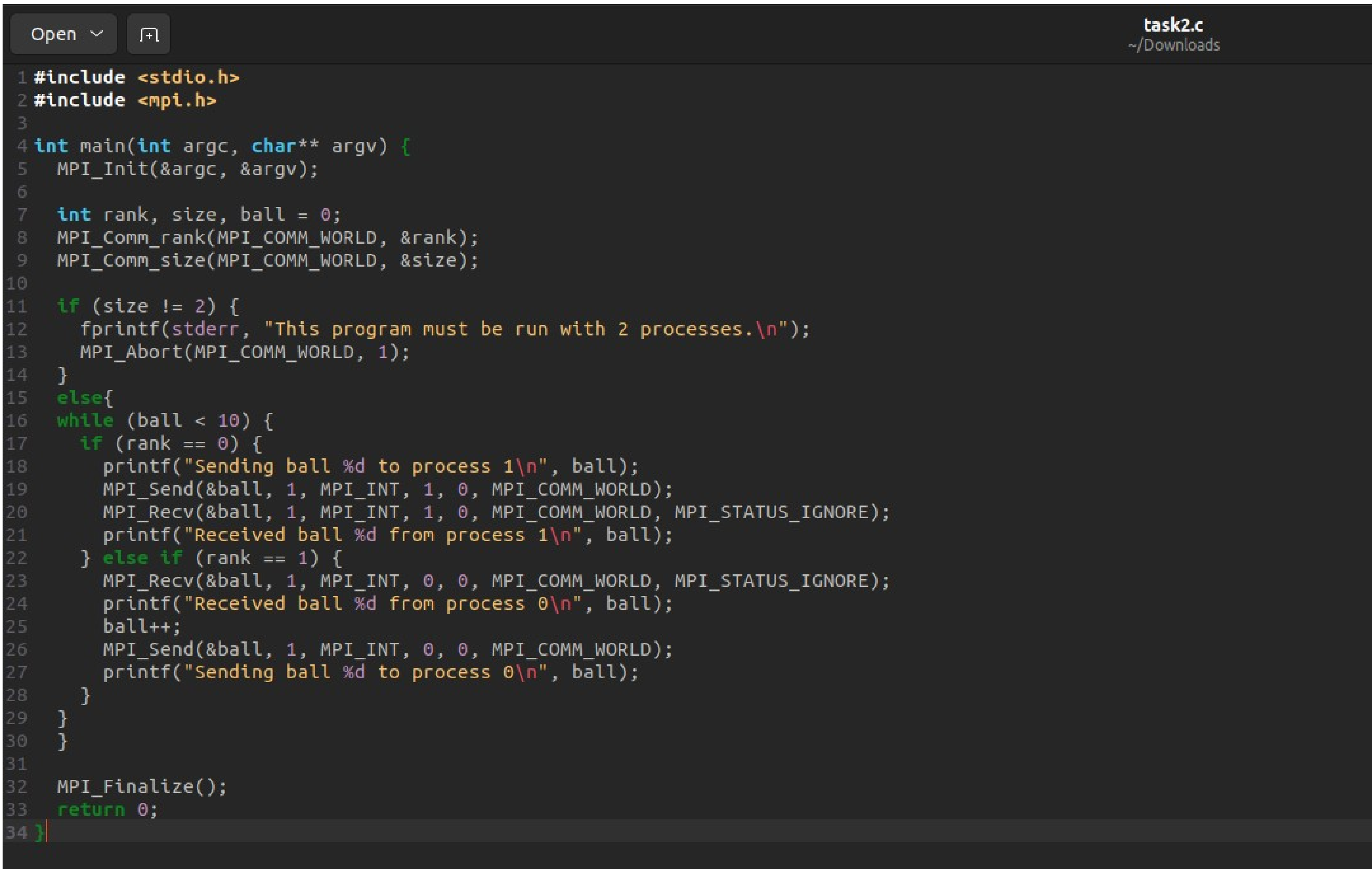
```
nayan@Nayan:~/Downloads$ mpicc task1.c -o task1
nayan@Nayan:~/Downloads$ mpiexec -n -4 ./task1
Total sum is 10000
nayan@Nayan:~/Downloads$
```

Explanation:

In this program the data is divided into chunks and distributed among multiple processes. Each process calculates the sum of its own chunk and sends the result back to the root process, which adds up the results from all processes to get the total sum of the array. Finally, the total sum is printed out.

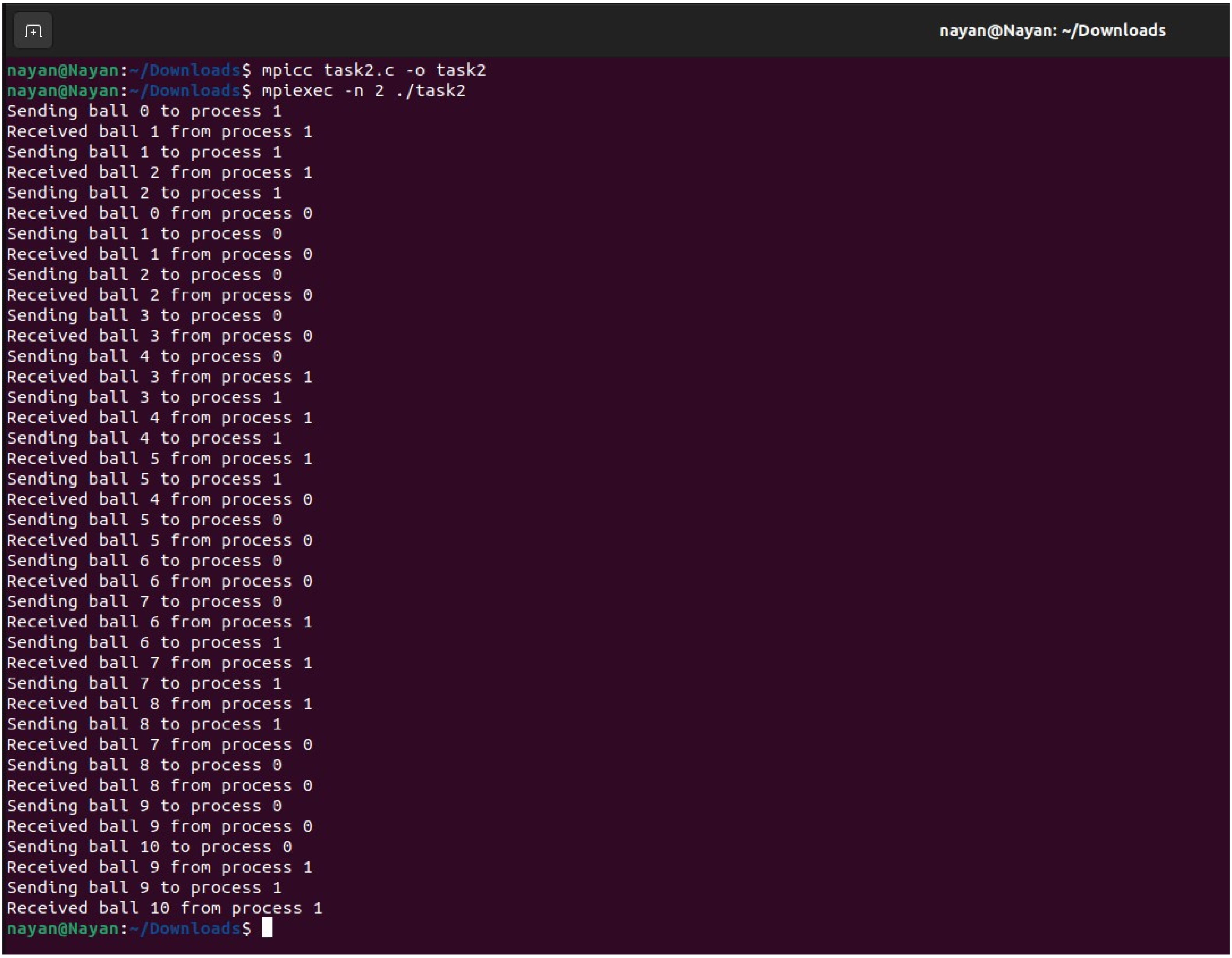
2. Write an MPI program called pingpong.c to run with exactly 2 processes. Process rank 0 is to send an integer variable called "ball" initialised with the value zero to Process rank 1. Process rank 1 will add 1 to the ball and send it back. This will repeat until the ball has a value of 10 in Process rank 0.

Code



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

int main(int argc, char** argv) {
  MPI_Init(&argc, &argv);

  int rank, size, ball = 0;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  if (size != 2) {
    fprintf(stderr, "This program must be run with 2 processes.\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
  }
  else{
  while (ball < 10) {
    if (rank == 0) {
      printf("Sending ball %d to process 1\n", ball);
      MPI_Send(&ball, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
      MPI_Recv(&ball, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("Received ball %d from process 1\n", ball);
    } else if (rank == 1) {
      MPI_Recv(&ball, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("Received ball %d from process 0\n", ball);
      ball++;
      MPI_Send(&ball, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
      printf("Sending ball %d to process 0\n", ball);
    }
  }
  }

  MPI_Finalize();
  return 0;
}
```

Output



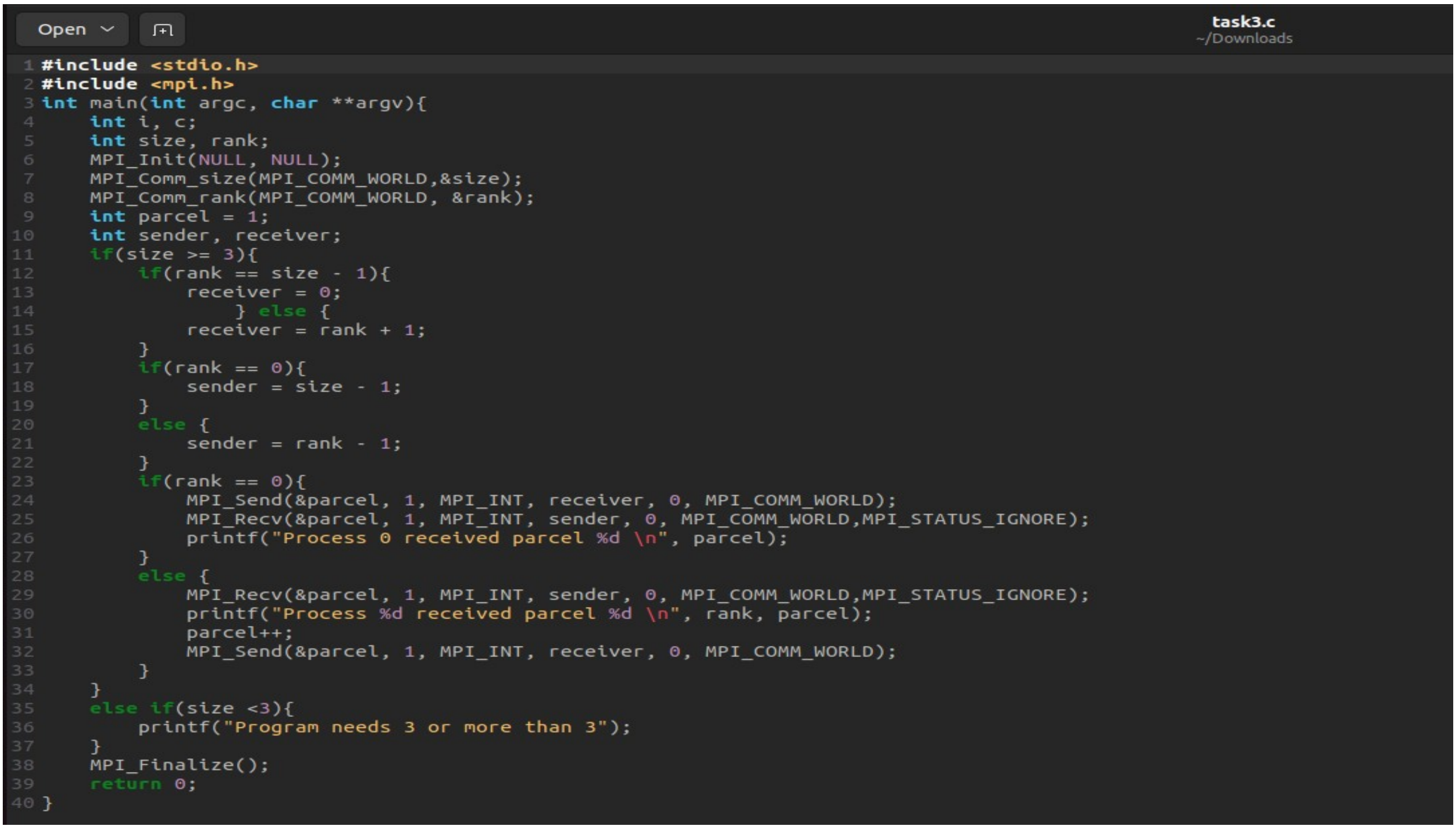
```
nayan@Nayan:~/Downloads$ mpicc task2.c -o task2
nayan@Nayan:~/Downloads$ mpiexec -n 2 ./task2
Sending ball 0 to process 1
Received ball 1 from process 1
Sending ball 1 to process 1
Received ball 2 from process 1
Sending ball 2 to process 1
Received ball 0 from process 0
Sending ball 1 to process 0
Received ball 1 from process 0
Sending ball 2 to process 0
Received ball 2 from process 0
Sending ball 3 to process 0
Received ball 3 from process 0
Sending ball 4 to process 0
Received ball 3 from process 1
Sending ball 3 to process 1
Received ball 4 from process 1
Sending ball 4 to process 1
Received ball 5 from process 1
Sending ball 5 to process 1
Received ball 4 from process 0
Sending ball 5 to process 0
Received ball 5 from process 0
Sending ball 6 to process 0
Received ball 6 from process 0
Sending ball 7 to process 0
Received ball 6 from process 1
Sending ball 6 to process 1
Received ball 7 from process 1
Sending ball 7 to process 1
Received ball 8 from process 1
Sending ball 8 to process 1
Received ball 7 from process 0
Sending ball 8 to process 0
Received ball 8 from process 0
Sending ball 9 to process 0
Received ball 9 from process 0
Sending ball 10 to process 0
Received ball 9 from process 1
Sending ball 9 to process 1
Received ball 10 from process 1
nayan@Nayan:~/Downloads$
```

Explanation:

In this program the code initializes MPI and gets the rank and size of the communicator. If the size is not 2, the program exits with an error message. The code then enters a loop where a ball is passed back and forth between the two processes.

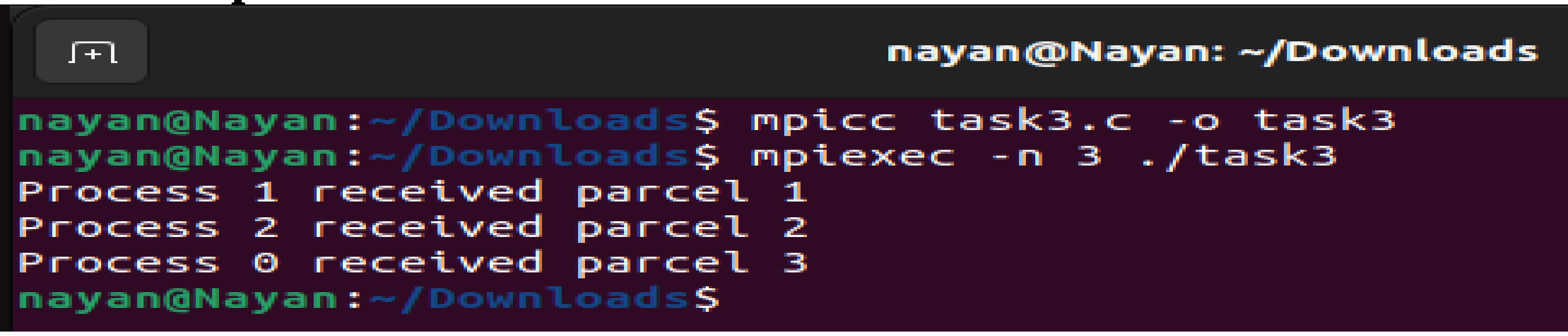
3. Write a "Pass-the-parcel" MPI program that will run with 3 or more nodes, such that Process rank 0 will send an integer variable call "parcel" initialised with 1, to Process rank 1 which will add 1 to the parcel and then send it to Process rank 2, and so on until the highest rank process will send it back to Process rank 0, at which point the parcel variable should contain the value of the number of nodes there are.

Code



```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv){
    int i, c;
    int size, rank;
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int parcel = 1;
    int sender, receiver;
    if(size >= 3){
        if(rank == size - 1){
            receiver = 0;
                } else {
            receiver = rank + 1;
        }
        if(rank == 0){
            sender = size - 1;
        }
        else {
            sender = rank - 1;
        }
        if(rank == 0){
            MPI_Send(&parcel, 1, MPI_INT, receiver, 0, MPI_COMM_WORLD);
            MPI_Recv(&parcel, 1, MPI_INT, sender, 0, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
            printf("Process 0 received parcel %d \n", parcel);
        }
        else {
            MPI_Recv(&parcel, 1, MPI_INT, sender, 0, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
            printf("Process %d received parcel %d \n", rank, parcel);
            parcel++;
            MPI_Send(&parcel, 1, MPI_INT, receiver, 0, MPI_COMM_WORLD);
        }
    }
    else if(size <3){
        printf("Program needs 3 or more than 3");
    }
    MPI_Finalize();
    return 0;
}
```

Output



```
nayan@Nayan:~/Downloads$ mpicc task3.c -o task3
nayan@Nayan:~/Downloads$ mpiexec -n 3 ./task3
Process 1 received parcel 1
Process 2 received parcel 2
Process 0 received parcel 3
nayan@Nayan:~/Downloads$
```

Explanation:

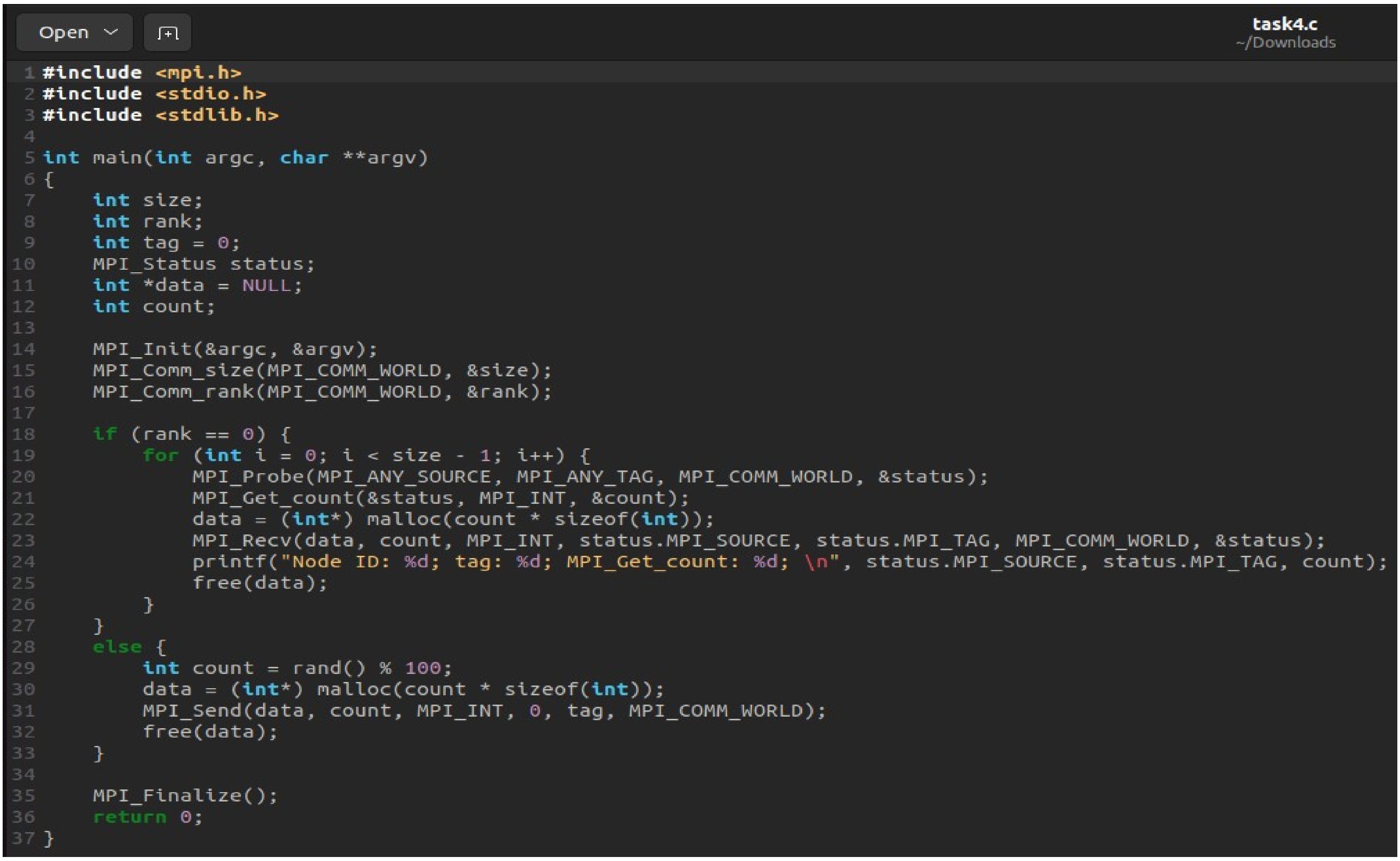
In this program a parcel is sent from one process to the next until it reaches the last process, which sends it back to the first. The program checks if there are at least three processes running, and if not, it prints an error message.

4. The following program has all the processes with rank greater than 0 send random amounts of data to Process rank 0 :

```
#include <mpi.h>

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

#define NUMDATA 1000

int main(int argc, char **argv)
{
  int size;
  int rank;
  int tag=0;
  int count;
  MPI_Status status;
  int data[NUMDATA];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0){
    for(int i = 0; i < size - 1; i++) {
   MPI_Recv(data, NUMDATA, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                     MPI_COMM_WORLD, &status);
      MPI_Get_count(&status, MPI_INT, &count);
      printf("Node ID: %d; tag: %d; MPI_Get_count: %d; \n",
             status.MPI_SOURCE, status.MPI_TAG, count);
    }
  }
  else {
    MPI_Send(data, rand()%100, MPI_INT, 0, tag, MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return 0;
}
```

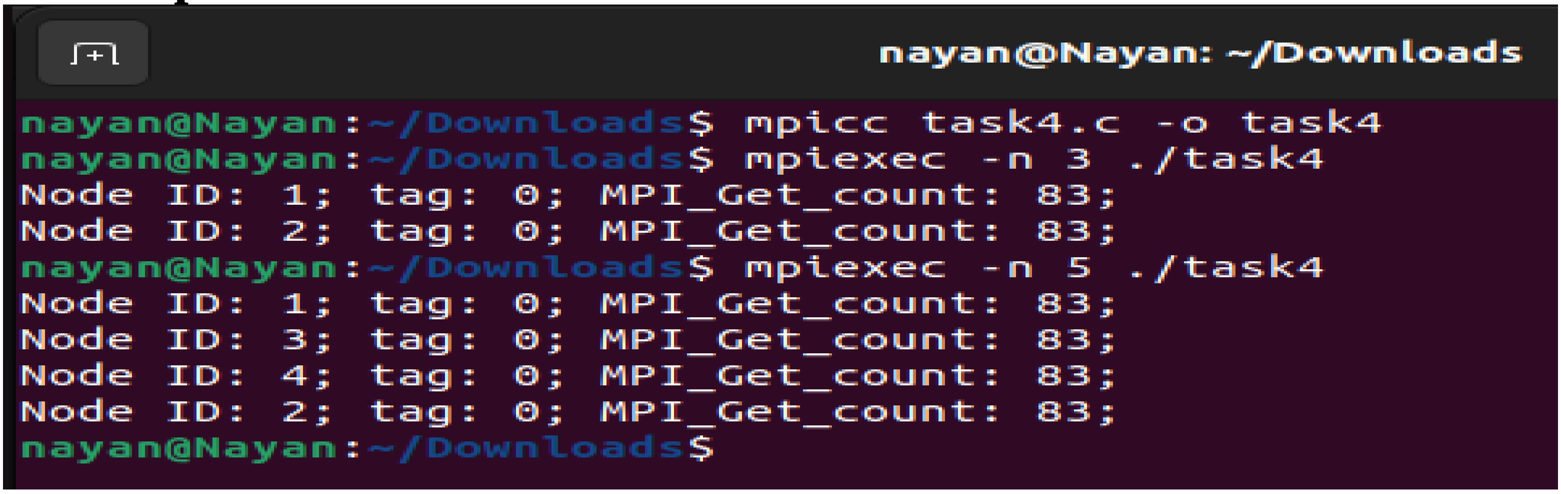
Modify it so that it doesn't use a fixed size data buffer but uses "malloc()" to allocate the correct amount of memory to use every time. Note this means that Process rank 0 will need to know what the amount of data it is expecting before it receives it.

Code



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

int main(int argc, char **argv)
{
    int size;
    int rank;
    int tag = 0;
    MPI_Status status;
    int *data = NULL;
    int count;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        for (int i = 0; i < size - 1; i++) {
            MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            MPI_Get_count(&status, MPI_INT, &count);
            data = (int*) malloc(count * sizeof(int));
            MPI_Recv(data, count, MPI_INT, status.MPI_SOURCE, status.MPI_TAG, MPI_COMM_WORLD, &status);
            printf("Node ID: %d; tag: %d; MPI_Get_count: %d; \n", status.MPI_SOURCE, status.MPI_TAG, count);
            free(data);
        }
    }
    else {
        int count = rand() % 100;
        data = (int*) malloc(count * sizeof(int));
        MPI_Send(data, count, MPI_INT, 0, tag, MPI_COMM_WORLD);
        free(data);
    }

    MPI_Finalize();
    return 0;
}
```

Output



```
nayan@Nayan:~/Downloads$ mpicc task4.c -o task4
nayan@Nayan:~/Downloads$ mpiexec -n 3 ./task4
Node ID: 1; tag: 0; MPI_Get_count: 83;
Node ID: 2; tag: 0; MPI_Get_count: 83;
nayan@Nayan:~/Downloads$ mpiexec -n 5 ./task4
Node ID: 1; tag: 0; MPI_Get_count: 83;
Node ID: 3; tag: 0; MPI_Get_count: 83;
Node ID: 4; tag: 0; MPI_Get_count: 83;
Node ID: 2; tag: 0; MPI_Get_count: 83;
nayan@Nayan:~/Downloads$
```

Explanation:

In this program each node either sends a message to node 0 or waits for a message from other nodes, and node 0 receives messages from all other nodes and prints some information about them. The messages are integer arrays with a randomly generated length between 0 and 99.

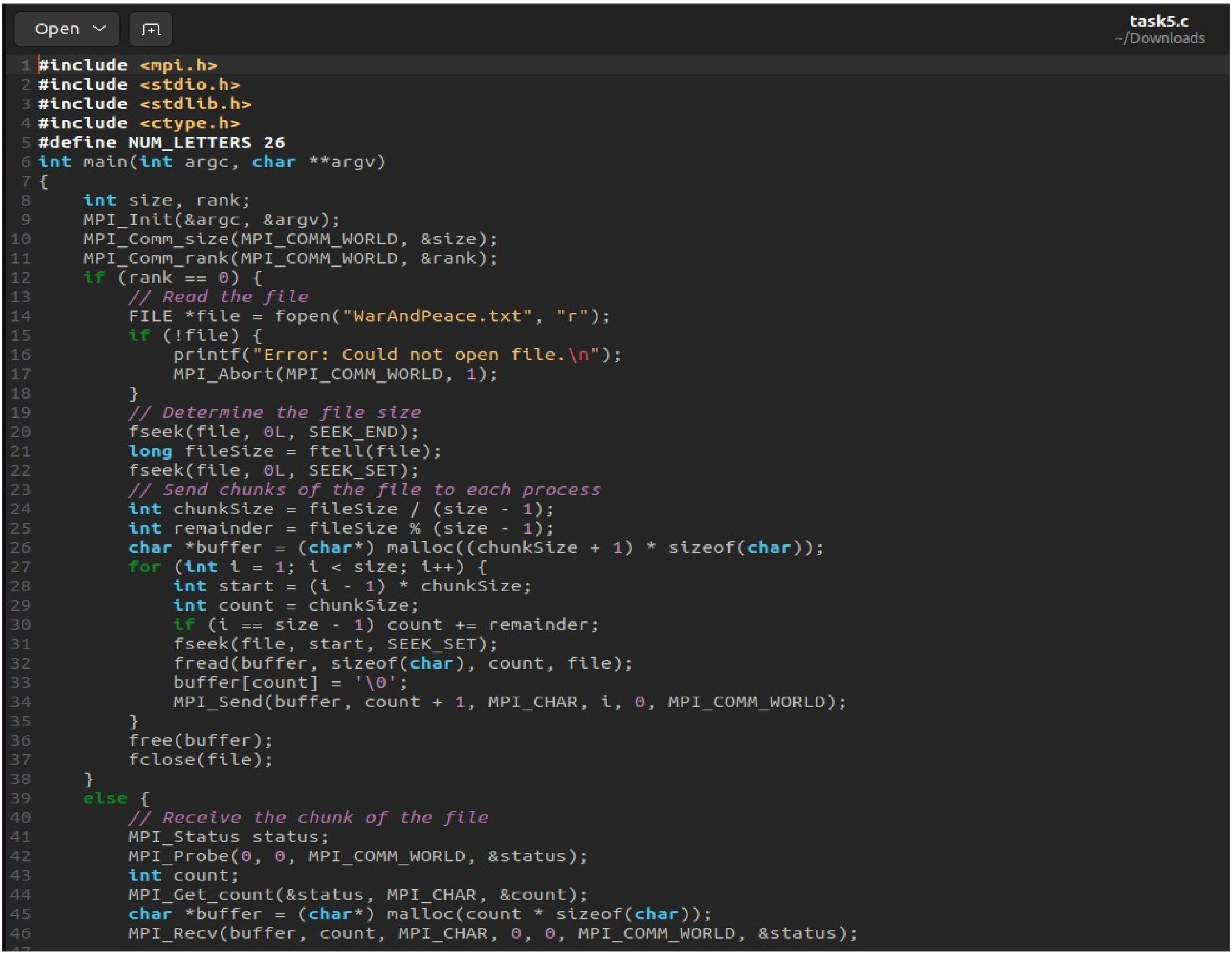
Assessed Workshop Task

5. The file "WarAndPeace.txt" on Canvas contains the entire text of the book "War and Peace" by Leo Tolstoy. Write an MPI program to count the number of times each letter of the alphabet occurs in the book. Count both the upper case and the lowercase as the same. Ignore any letter with accents such as " é " and so on.

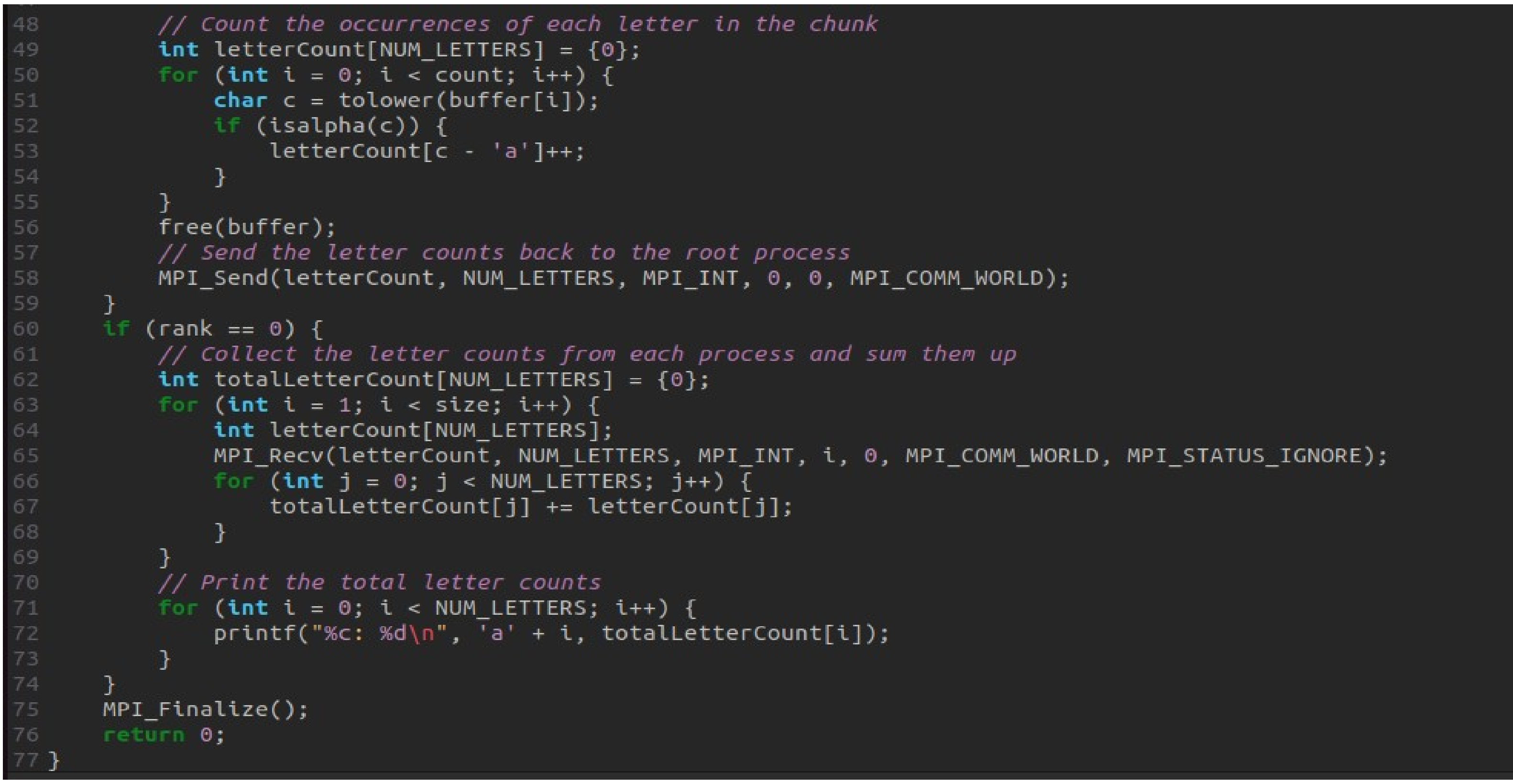
Your MPI program should work with any number of processes from 1 to 100 processes. Only Process rank 0 (zero) should read in the file and send the **appropriate** chunk of file to each other process. The other processes should not read in the file.

You should submit this program as "workshoptask1.c" as part of your final portfolio submission. You can also upload it to the formative submission point for formative feedback.

Code

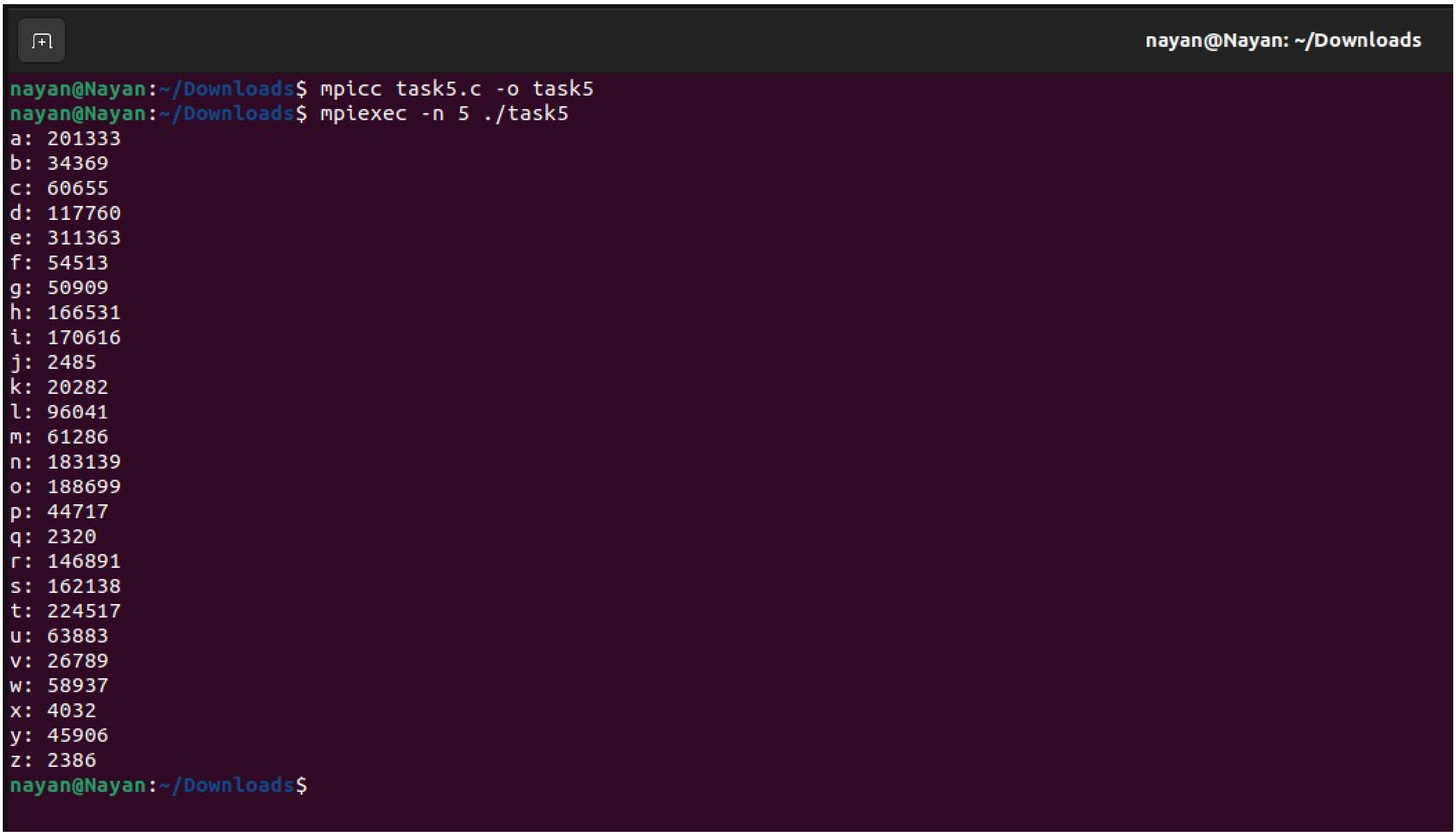


```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#define NUM_LETTERS 26
int main(int argc, char **argv)
{
    int size, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        // Read the file
        FILE *file = fopen("WarAndPeace.txt", "r");
        if (!file) {
            printf("Error: Could not open file.\n");
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
        // Determine the file size
        fseek(file, 0L, SEEK_END);
        long fileSize = ftell(file);
        fseek(file, 0L, SEEK_SET);
        // Send chunks of the file to each process
        int chunkSize = fileSize / (size - 1);
        int remainder = fileSize % (size - 1);
        char *buffer = (char*) malloc((chunkSize + 1) * sizeof(char));
        for (int i = 1; i < size; i++) {
            int start = (i - 1) * chunkSize;
            int count = chunkSize;
            if (i == size - 1) count += remainder;
            fseek(file, start, SEEK_SET);
            fread(buffer, sizeof(char), count, file);
            buffer[count] = '\0';
            MPI_Send(buffer, count + 1, MPI_CHAR, i, 0, MPI_COMM_WORLD);
        }
        free(buffer);
        fclose(file);
    }
    else {
        // Receive the chunk of the file
        MPI_Status status;
        MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
        int count;
        MPI_Get_count(&status, MPI_CHAR, &count);
        char *buffer = (char*) malloc(count * sizeof(char));
        MPI_Recv(buffer, count, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &status);
```



```c
        // Count the occurrences of each letter in the chunk
        int letterCount[NUM_LETTERS] = {0};
        for (int i = 0; i < count; i++) {
            char c = tolower(buffer[i]);
            if (isalpha(c)) {
                letterCount[c - 'a']++;
            }
        }
        free(buffer);
        // Send the letter counts back to the root process
        MPI_Send(letterCount, NUM_LETTERS, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    if (rank == 0) {
        // Collect the letter counts from each process and sum them up
        int totalLetterCount[NUM_LETTERS] = {0};
        for (int i = 1; i < size; i++) {
            int letterCount[NUM_LETTERS];
            MPI_Recv(letterCount, NUM_LETTERS, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            for (int j = 0; j < NUM_LETTERS; j++) {
                totalLetterCount[j] += letterCount[j];
            }
        }
        // Print the total letter counts
        for (int i = 0; i < NUM_LETTERS; i++) {
            printf("%c: %d\n", 'a' + i, totalLetterCount[i]);
        }
    }
    MPI_Finalize();
    return 0;
}
```

Output



```
nayan@Nayan:~/Downloads$ mpicc task5.c -o task5
nayan@Nayan:~/Downloads$ mpiexec -n 5 ./task5
a: 201333
b: 34369
c: 60655
d: 117760
e: 311363
f: 54513
g: 50909
h: 166531
i: 170616
j: 2485
k: 20282
l: 96041
m: 61286
n: 183139
o: 188699
p: 44717
q: 2320
r: 146891
s: 162138
t: 224517
u: 63883
v: 26789
w: 58937
x: 4032
y: 45906
z: 2386
nayan@Nayan:~/Downloads$
```

Explanation:

In this program the file is divided into chunks and sent to each process (except the root process) for counting. The root process receives the letter counts from each process and sums them up to obtain the total letter counts. The program then prints the results.