CMPE 300 ANALYSIS OF ALGORITHMS

2020 Spring - MPI Programming Application Project

Submitted by İsmet Sarı

İsmet Sarı
2016400324

June 3, 2020

Contents

1 Introduction

Message Passing Interface (MPI) is a standardized and portable message-passing standard designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures. If you have enough processor to solve a problem parallel this technique is the elegant to reach out the fastest result. In industry the problem is divided into as many part as processor number, assign each of them a part of the problem and create a communication between these processor to get the flow of information. This communication interface is called MPI model. In many big companies such as IBM or research platforms such as Argonne National Laboratory, MPI have been used. Early 90s, IBM's super computers and RAMs were designed by the implementation of MPI. In this project we have two intervals and we are assigned to find the all Armstrong numbers between these two intervals so we divide the interval into the number of processor, which is given by user and assign these parts to its regarding processor and create a communication built upon send and receive operations among processor.

2 Program Interface

You can execute this project in a system that has MPI environment so before executing this project you need to install MPI in your operation system. I preferred to install in Ubuntu but you can do it in any operation system. To install in Ubuntu, follow these steps.

- First Download latest stable release of open MPI from here.
- Type these commands to build Open MPI with default settings (if want to change defaults, find and read the Open MPI installation guide)
- and then to extract these file type tar -xvf openmpi-1.4.3.tar.gz command.
- execute configure file by ./configure and finally make all install

To complite the program you can just type make in path the project file called ismet.c reside or type mpicc -g ismet.c -o ismet on terminal. After applying these step your system is ready to execute my project.

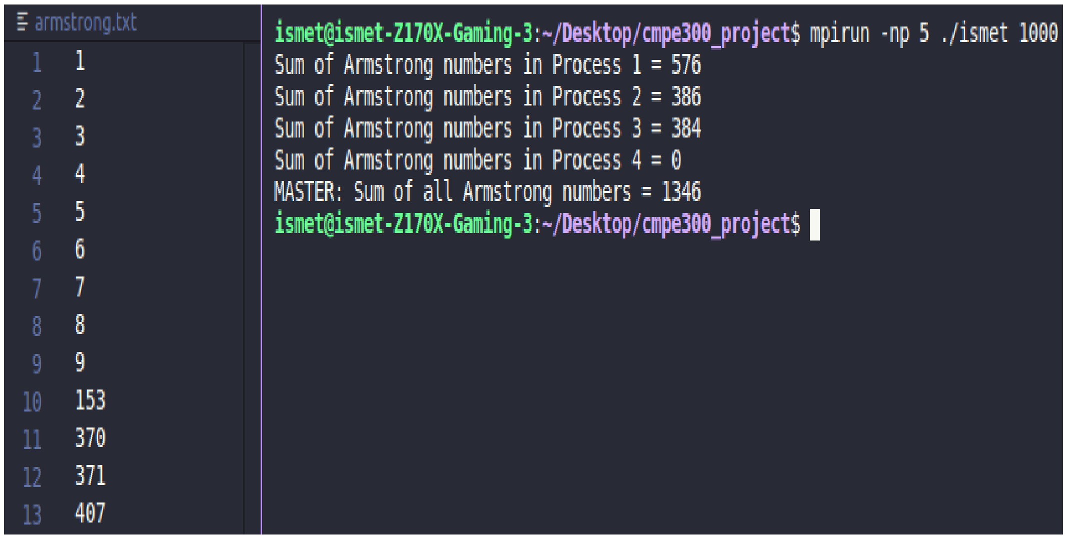
3 Program Execution

3.1 Input - Output

After installing and compiling steps, to run the program you need to use mpirun or mpiexec command in a such a way that mpirun (or mpiexec) -np 11 ./ismet 10000. In this command -np decide how many processor is going to be allocated and 10000 is the upper bound of the interval that we are going to find Armstrong numbers between them. Of course you can change these numbers.

The program output on two platform. One of them is on the terminal. Every processor including master print out the total of Armstrong numbers between its current interval. In the example above we send the numbers between 0-10000 to the program and allocate 11 processors in such a way that 1 is master processor 10 is worker processors. Every worker processor search 10000/10 = 1000 numbers and tries to find Armstrong numbers and print out the total of them on the terminal. Master processor takes all the Armstrong numbers from last processor and prints out the total of all the Armstrong numbers on the terminal and also sort them up create a file called Armstrong.txt and writes the sorted list on this file.

You can see the input and output format from the picture below.



armstrong.txt

```
1
2
3
4
5
6
7
8
9
153
370
371
407
```

```
ismet@ismet-Z170X-Gaming-3:~/Desktop/cmpe300_project$ mpirun -np 5 ./ismet 1000
Sum of Armstrong numbers in Process 1 = 576
Sum of Armstrong numbers in Process 2 = 386
Sum of Armstrong numbers in Process 3 = 384
Sum of Armstrong numbers in Process 4 = 0
MASTER: Sum of all Armstrong numbers = 1346
ismet@ismet-Z170X-Gaming-3:~/Desktop/cmpe300_project$
```

3.2 Program Structure

We need to first include all the necessary libraries and define some variables that we are going to use.

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

#define MASTER 0 /* taskid of first process */
#define indexmsg 1
#define arraymsg 2
```

In main function we first create a data array which is dynamically allocated based on the input size and this data array will be filled up the all numbers between given intervals and shared by all processors. Each processor is going to be responsible of one part of this array. After allocating this array we need to start the over all processor by MPI_Init(NULL, NULL); After this step we have as many processors as the user give available.

```
// Find out rank, size
int numtasks, taskid;
MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
int numworkers;
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

numworkers = numtasks - 1;
int chunksize = (ARRAYSIZE / numworkers);
```

Then we need to define who we are taskid and numtask variables are defined and assinged by MPI_Command functions. If the taskid is 0 then we know that we are in master processor if it is higher then 0 then we are currently in worker processor. The number of numworker variable might be defined in master processor and send this number to each worker processors to callculate the chunksize.

```
numworkers = numtasks - 1;
int chunksize = (ARRAYSIZE / numworkers);

if (taskid == MASTER)
{

    for (int i = 0; i < ARRAYSIZE; i++)
        data[i] = i + 1;
    shuffle(data, ARRAYSIZE);
    int index = 0;

    for (int i = 1; i <= numworkers; i++)
    {

        MPI_Send(&data[index], chunksize, MPI_INT, i, arraymsg,
                 MPI_COMM_WORLD);
        index = index + chunksize;
    }
```

As you can see above in main function we decide the number of workers and divide the interval by this number the get the part that each processor is working on. We are told that the master create the all numbers between the given interval, assign these number into data array and shuffle them so the shuffle operation is handled by this program below.

```
void shuffle(int *array, size_t n)
{
    if (n > 1)
    {
        size_t i;
        for (i = 0; i < n - 1; i++)
        {
            size_t j = i + rand() / (RAND_MAX / (n - i) + 1);
            int t = array[j];
            array[j] = array[i];
            array[i] = t;
        }
    }
}
```

After shuffling we are sending each worker processors a part of the data array. As you can see the tag of the sending operation is arraymsg which is 2 defined above. After master processor do these steps it is going to wait until last processors return some values.

Workers processors first waits for master processor to send their regarding part of the array and they receive this part by MPI receive as you can see below worker processors receive a list of numbers with a length of chunk-size which is calculated in main function above.

```
if (taskid > MASTER)
{

    int source = MASTER;
    int index = (taskid - 1) * chunksize;

    int local_total = 0;
    int local_size_of_amstrong_numbers = 0;
    int limit = 0;
    int global_total_of_amstrong_numbers = 0;
    if (taskid == 1)
        limit = chunksize / 2;
    else
        limit = chunksize / 3;

    int *local_amstrong_numbers;

    local_amstrong_numbers = (int *)calloc(limit, sizeof(int));

    MPI_Recv(&data[index], chunksize, MPI_INT, source, arraymsg,
             MPI_COMM_WORLD, &status);
```

We define some variables. first we need to understand which index of the data array we are going to start to search. So we subtract 1 from taskid and multiply this number by chunksize to get this. and define a limit variable. For the first worker process I behave very generous by assigning half of the chunksize. If we have 100 number to search we can have at most 20 Armstrong number so it could be better to assign on fifth of chunksize. After all based on this limit size we are construction the local_amstrong_numbers array to store Armstrong numbers which are found in current processor and finally receive the regarding part of the data array.

```
 for (int i = index; i < index + chunksize; i++)
 {

     int originalNumber;
     int number = data[i];
     int count = 0;

     originalNumber = number;

     // number of digits calculation
     while (originalNumber != 0)
     {
         originalNumber /= 10;
         ++count;
     }

     originalNumber = number;
     int rem = 0;
     int result = 0;

     // result contains sum of nth power of individual digits
     while (originalNumber != 0)
     {
         rem = originalNumber % 10;
         result += pow(rem, count);
         originalNumber /= 10;
     }
    ,

    // check if number is equal to the sum of nth power of individual digits
    if ((int)result == number)
    {
        if (limit == local_size_of_amstrong_numbers)
        {
            limit = limit * 2;
            local_amstrong_numbers = realloc(local_amstrong_numbers, limit * sizeof(int));
        }

        local_amstrong_numbers[local_size_of_amstrong_numbers] = number;
        local_size_of_amstrong_numbers + = 1;
        local_total = local_total + number;
    }
}

printf("Sum of Armstrong numbers in Process %d = %d\n", taskid, local_total);

global_total_of_amstrong_numbers + = local_total;
```

After deciding the index and receiving the array we traverse the array and check if the number is Armstrong number or not. If the number is amstrong number them we add this number to local_total variable and local_amstrong_numbers array and increase the local_size_of_amstrong_numbers by 1. We print the local total number of Armstrong numbers on terminal. After all we add the total number of Armstrong numbers and global total number of Armstrong number. There is one if statement in assigning operation since if the data array size is 20 and number of worker processors is 4 then we assign each processors 5 number to search bu we set the limit as 5/2=2. Then we cannot assign local array the 3 4 5 even though these are Armstrong number and index error occurs. To overcome this problem we reallocate the local array and double the size.

```
MPI_Send(&local_size_of_amstrong_numbers, 1, MPI_INT, MASTER, 3, MPI_COMM_WORLD);
MPI_Send(&local_amstrong_numbers[0], local_size_of_amstrong_numbers, MPI_INT, MASTER, 4, MPI_COMM_WORLD);
```

Each processor sends an array which is filled up all the Armstrong number found in its regarding part of the data array to the master processor. To send properly we sent first the size of this array with tag 3 and after that we send actual array with tag 4. The communication between master and workers is specified by these tags. As sending from master to workers we use tag-1(for int),tag-2(for arrays) and receiving from workers we use tag-3(for int) tag-4(for array).

```
int global_total_of_all_amstrong_numbers = 0;

int size_of_all_amstrong_numbers = 0;
int *global_total_array;
global_total_array = (int *)calloc((ARRAYSIZE / 5), sizeof(int));
for (int i = 1; i <= numworkers; i++)
{
    int size = 0;
    MPI_Recv(&size, 1, MPI_INT, i, 3, MPI_COMM_WORLD,
             &status);

    int *from_worker_process;
    from_worker_process = (int *)calloc(size, sizeof(int));
    MPI_Recv(&from_worker_process[0], size, MPI_INT, i, 4, MPI_COMM_WORLD,
             &status);
    for (int k = 0; k < size; k++)
    {
        global_total_array[size_of_all_amstrong_numbers] = from_worker_process[k];
        size_of_all_amstrong_numbers += 1;
    }
}
```

Master processors takes the arrays filled up Armstrong numbers from worker processors in a way that we define a for loop and start receiving. First receive the size of the array and construct this array dynamically based on this size. After taking this array we iterate this array and takes all its values to our all Armstrong number array.

```
if (taskid != numworkers)
{
    MPI_Send(&global_total_of_amstrong_numbers, 1, MPI_INT, taskid + 1, 11, MPI_COMM_WORLD);
}
```

```
if (taskid != 1)
    MPI_Recv(&global_total_of_amstrong_numbers, 1, MPI_INT, taskid - 1, 11,
             MPI_COMM_WORLD, &status);
```

Each processor send the total number of Armstrong number up to its state to next processor. For instance processor 2 send total number of Armstrong number that processor 1 and processor 1 have found to processor 3 and processor 3 add its local value and this value and send to processor 4. Each worker processor except processor 1 receive this number before starting to traverse its regarding part of data array. We send and receive this integer value by tag-11.

```
else if (taskid == numworkers)
{

    MPI_Send(&global_total_of_amstrong_numbers, 1, MPI_INT, MASTER, 5, MPI_COMM_WORLD);
}
```

However processor 10 will not send any other worker processor instead it will send the total value to master processor

```
MPI_Recv(&global_total_of_all_amstrong_numbers, 1, MPI_INT, numworkers, 5, MPI_COMM_WORLD,
         &status);


printf("MASTER: Sum of all Armstrong numbers = %d \n", global_total_of_all_amstrong_numbers);
```

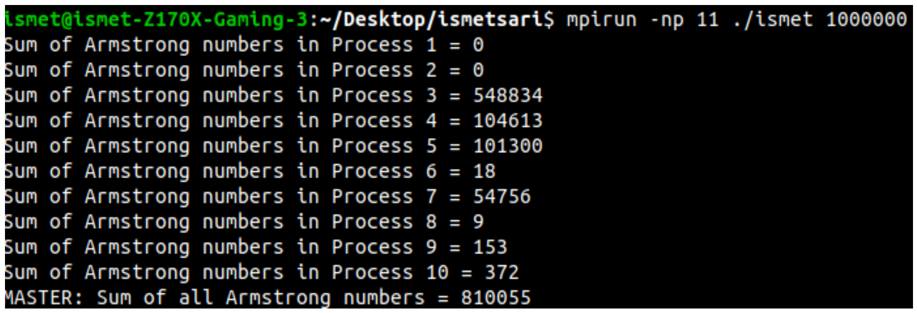
And master take this value and prints that value out on terminal.

```
quickSort(global_total_array, 0, size_of_all_amstrong_numbers - 1);
FILE *fptr;
fptr = fopen("armstrong.txt", "w+");
int armstrong_number = 0;
for (int i = 0; i < size_of_all_amstrong_numbers; i++)
{
    armstrong_number = global_total_array[i];
    fprintf(fptr, "%d\n", armstrong_number);
}
```

Last operator that the master processor need to do is to sort all these numbers and write on a file called armstrong.txt.

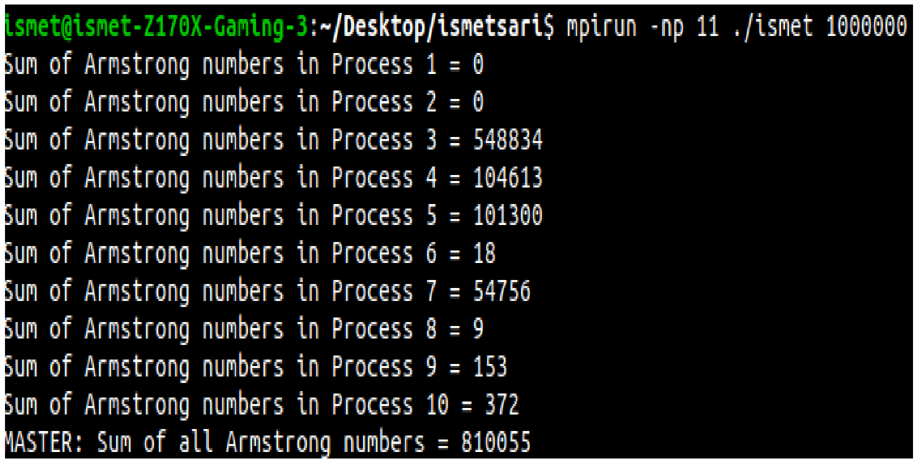
4 Examples

You can see some examples below.



```
ismet@ismet-Z170X-Gaming-3:~/Desktop/ismetsari$ mpirun -np 11 ./ismet 1000000
Sum of Armstrong numbers in Process 1 = 0
Sum of Armstrong numbers in Process 2 = 0
Sum of Armstrong numbers in Process 3 = 548834
Sum of Armstrong numbers in Process 4 = 104613
Sum of Armstrong numbers in Process 5 = 101300
Sum of Armstrong numbers in Process 6 = 18
Sum of Armstrong numbers in Process 7 = 54756
Sum of Armstrong numbers in Process 8 = 9
Sum of Armstrong numbers in Process 9 = 153
Sum of Armstrong numbers in Process 10 = 372
MASTER: Sum of all Armstrong numbers = 810055
```

As you can see the shell command above we allocate 11 processor and try to find all Armstrong numbers between 1000000 and 1.



```
ismet@ismet-Z170X-Gaming-3:~/Desktop/ismetsari$ mpirun -np 11 ./ismet 1000000
Sum of Armstrong numbers in Process 1 = 0
Sum of Armstrong numbers in Process 2 = 0
Sum of Armstrong numbers in Process 3 = 548834
Sum of Armstrong numbers in Process 4 = 104613
Sum of Armstrong numbers in Process 5 = 101300
Sum of Armstrong numbers in Process 6 = 18
Sum of Armstrong numbers in Process 7 = 54756
Sum of Armstrong numbers in Process 8 = 9
Sum of Armstrong numbers in Process 9 = 153
Sum of Armstrong numbers in Process 10 = 372
MASTER: Sum of all Armstrong numbers = 810055
```

As you can see the shell command above we allocate 5 processor to find all Armstrong numbers between 1000000 and 1. and between 100000 and 1.

5 Improvements and Extensions

Implementing this project I should have spent more time to specify the limit variable which we use to create dynamically allocated array called local_amstrong_numbers. For the first worker processor we allocate half of chunksize for other worker processors we allocate one of third of chunksize. Here we consider huge intervals and small number of processors. Chunk-size is very large but we expect to find one or two Armstrong number so the limit need to be very small but in any case we have a lot of space to store. This algorithm is not space efficient.

6 Difficulties Encountered

The source code was implemented in C language so as you expect you are doing everything yourself. I have dealt with some signal 6 and signal 11 errors during implementation. Allocating and reallocating parts were struggling parts.

7 Conclusion

In this project we implement try to implement a program that find the all Armstrong number between an interval using MPI. It is important for all of us to learn the basic of parallelism and MPI since all the systems around us were designed based on this approaches.

8 References

References

[1] Quick-Sort algorithm sort used to sort the returned Armstrong number in master array *https://www.geeksforgeeks.org/quick-sort/*. (Website)

[2] Shuffle array algorithm used to shuffle the data array before sending to processors. *https://stackoverflow.com/questions/6127503/shuffle-array-in-c*. (Website)

Appendices

- **Source Code**

```
/*Student Name: İsmet Sarı
Student Number: 2016400324
Compile Status: Compiling
Program Status: Working
*/
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define MASTER 0 /* taskid of first process */
#define indexmsg 1
#define arraymsg 2

void shuffle(int *array, size_t n)
{
    if (n > 1)
    {
        size_t i;
        for (i = 0; i < n - 1; i++)
        {
            size_t j = i + rand() / (RAND_MAX / (n - i) + 1);
            int t = array[j];
            array[j] = array[i];
            array[i] = t;
        }
    }
}
void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
int partition(int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1);     // Index of smaller element

    for (int j = low; j <= high - 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
 arr[] --> Array to be sorted,
  low  --> Starting index,
  high  --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

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

    // Initialize the MPI environment
    MPI_Status status;

    int ARRAYSIZE = atoi(argv[1]);
    int *data;
    data = (int *)calloc(ARRAYSIZE, sizeof(int));
    MPI_Init(NULL, NULL);
    // Find out rank, size
    int numtasks, taskid;
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
    int numworkers;
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

    numworkers = numtasks - 1;
    int chunksize = (ARRAYSIZE / numworkers);

    if (taskid == MASTER)
    {

        for (int i = 0; i < ARRAYSIZE; i++)
            data[i] = i + 1;
        shuffle(data, ARRAYSIZE);
        int index = 0;

        for (int i = 1; i <= numworkers; i++)
        {

            MPI_Send(&data[index], chunksize, MPI_INT, i, arraymsg,
                     MPI_COMM_WORLD);
            index = index + chunksize;
        }

        int global_total_of_all_amstrong_numbers = 0;

        int size_of_all_amstrong_numbers = 0;
        int *global_total_array;
        global_total_array = (int *)calloc((ARRAYSIZE / 5), sizeof(int));
        for (int i = 1; i <= numworkers; i++)
        {
            int size = 0;
            MPI_Recv(&size, 1, MPI_INT, i, 3, MPI_COMM_WORLD,
                     &status);

            int *from_worker_process;
            from_worker_process = (int *)calloc(size, sizeof(int));
            MPI_Recv(&from_worker_process[0], size, MPI_INT, i, 4, MPI_COMM_WORLD,
                     &status);
            for (int k = 0; k < size; k++)
            {
                global_total_array[size_of_all_amstrong_numbers] = from_worker_process[k];
                size_of_all_amstrong_numbers += 1;
            }
        }

        MPI_Recv(&global_total_of_all_amstrong_numbers, 1, MPI_INT, numworkers, 5,
MPI_COMM_WORLD,
                 &status);

        quickSort(global_total_array, 0, size_of_all_amstrong_numbers - 1);
        FILE *fptr;
        fptr = fopen("armstrong.txt", "w+");
        int armstrong_number = 0;
        for (int i = 0; i < size_of_all_amstrong_numbers; i++)
        {
            armstrong_number = global_total_array[i];
            fprintf(fptr, "%d\n", armstrong_number);
        }

        printf("MASTER: Sum of all Armstrong numbers = %d \n",
global_total_of_all_amstrong_numbers);
    }

    if (taskid > MASTER)
    {

        int source = MASTER;
        int index = (taskid - 1) * chunksize;

        int local_total = 0;
        int local_size_of_amstrong_numbers = 0;
        int limit = 0;
        int global_total_of_amstrong_numbers = 0;
        if (taskid == 1)
            limit = chunksize / 2;
        else
            limit = chunksize / 3;

        int *local_amstrong_numbers;

        local_amstrong_numbers = (int *)calloc(limit, sizeof(int));

        MPI_Recv(&data[index], chunksize, MPI_INT, source, arraymsg,
                 MPI_COMM_WORLD, &status);
        if (taskid != 1)
            MPI_Recv(&global_total_of_amstrong_numbers, 1, MPI_INT, taskid - 1, 11,
                MPI_COMM_WORLD, &status);

        for (int i = index; i < index + chunksize; i++)
        {

            int originalNumber;
            int number = data[i];
            int count = 0;

            originalNumber = number;

            // number of digits calculation
            while (originalNumber != 0)
            {
                originalNumber /= 10;
                ++count;
            }

            originalNumber = number;
            int rem = 0;
            int result = 0;

            // result contains sum of nth power of individual digits
            while (originalNumber != 0)
            {
                rem = originalNumber % 10;
                result += pow(rem, count);
                originalNumber /= 10;
            }

            // check if number is equal to the sum of nth power of individual digits
            if ((int)result == number)
            {
                if (limit == local_size_of_amstrong_numbers)
                {
                    limit = limit * 2;
                    local_amstrong_numbers = realloc(local_amstrong_numbers, limit * sizeof(int));
                }

                local_amstrong_numbers[local_size_of_amstrong_numbers] = number;
                local_size_of_amstrong_numbers + = 1;
                local_total = local_total + number;
            }
        }

        printf("Sum of Armstrong numbers in Process %d = %d\n", taskid, local_total);

        global_total_of_amstrong_numbers + = local_total;

        MPI_Send(&local_size_of_amstrong_numbers, 1, MPI_INT, MASTER, 3,
MPI_COMM_WORLD);
        MPI_Send(&local_amstrong_numbers[0], local_size_of_amstrong_numbers, MPI_INT,
MASTER, 4, MPI_COMM_WORLD);

        if (taskid != numworkers)
        {
            MPI_Send(&global_total_of_amstrong_numbers, 1, MPI_INT, taskid + 1, 11,
MPI_COMM_WORLD);
        }
        else if (taskid == numworkers)
        {

            MPI_Send(&global_total_of_amstrong_numbers, 1, MPI_INT, MASTER, 5,
MPI_COMM_WORLD);
        }
    }

    MPI_Finalize();

    return 0;
}
```