

Homework #2

ADD\_NUM\_MPI

cpe 512

Kyle Ray

August 31, 2017

Contents

[Add\_Num\_MPI\_rev1 2](#_Toc493706620)

[Source Code 2](#_Toc493706621)

[Output 8](#_Toc493706622)

[Add\_Num\_MPI\_rev2 9](#_Toc493706623)

[Source Code 10](#_Toc493706624)

[Output 16](#_Toc493706625)

[Answers 17](#_Toc493706626)

[Add\_Num\_MPI\_rev3 18](#_Toc493706627)

[Source Code 18](#_Toc493706628)

[Output 25](#_Toc493706629)

[Answers 26](#_Toc493706630)

# Add\_Num\_MPI\_rev1

## Source Code

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Summation of a Sequence of Numbers Program -- MPI version \*/

/\* September 2017 -- B. Earl Wells -- University of Alabama \*/

/\* in Huntsville \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

This program illustrates the basic concepts of SPMD programming using

MPI. The program represents a common example that is used often in

the CPE 412/512 text, the distributed addition of a sequence of numbers.

The program is written in such a way that it is assumed that the

sequence of numbers is first read from a central source (in this case

a data file) by a single MPI process (the root process) and then

partitioned into equal groups with each part being distributed

(scattered) to the local memory domains of the other MPI

processes in the system. After which each process computes its own

partial sum of the data that is in its domain and then sends this

value to the memory domain of the root process which then

adds the partial sums together (reduce operation) and then

outputs this sum to the screen.

The following is a simplified version of the program which you will

be asked to augment in a number of ways in future homework

assignments. Use the dmc.asc.edu or the Jetson Cluster system for

these assignments.

Notes: This implementation utilizes a minimum set of MPI function

call that include MPI\_Init, MPI\_Finalize, MPI\_Comm\_size,

MPI\_Comm\_rank, MPI\_Send, and MPI\_Recv.

MPI\_Abort is also used to illustrate its functionality.

To compile type:

module load openmpi

mpic++ add\_num\_MPI.cpp -o add\_num\_MPI

To execute:

mpiexec -np [num MPI process] add\_num\_MPI [num of numbers]

EDIT: Kyle Ray

CPE\_512 Intro to Parallel Programming

Homework #2

September 21, 2017

Addition:

Allow each process to compute a local min and max.

Report these to the root and have the root find the global min and max

Report this to the user.

\*/

using namespace std;

#include <iostream>

#include <iomanip>

#include <sstream>

#include <stdlib.h>

#include <mpi.h> /\* MPI Prototype Header Files \*/

// Defines so that I can compile the code in visual studio

//#define srand48(s) srand(s)

//#define drand48() (((double)rand())/((double)RAND\_MAX))

#define SEED 2397 /\* random number seed \*/

#define MAX\_VALUE 100.0 /\* maximum value of any number in list \*/

#define MIN\_VALUE -50.0 /\* minimum value of any number in list \*/

/\*

ONE-TO-ALL BROADCAST COMMUNICATION ROUTINE

Routine to transfer from the root MPI process the value of

the 'int\_num' parameter to all other MPI processes in the system.

\*/

void broadcast\_int(int \*int\_num, int root, int rank, int numtasks) {

MPI\_Status status;

int type = 123;

// root send value of int\_num to each of the other processes

// using a locally blocking point-to-point send

if (rank == root) {

for (int mpitask = 0; mpitask < numtasks; mpitask++) {

if (mpitask != root) {

MPI\_Send(int\_num, 1, MPI\_INT,

mpitask, type, MPI\_COMM\_WORLD);

}

}

}

// if not root process execute a blocking point-to-point receive

// with the source being to root process and direct this data to

// the local copy of 'int\_num'

else {

MPI\_Recv(int\_num, 1, MPI\_INT,

root, type, MPI\_COMM\_WORLD, &status);

}

}

/\*

Routine to retrieve the data size of the numbers array from the

command line or get this number by prompting the user for the

information. Note: command line values are sent to ALL MPI processes

by the MPI environment.

\*/

int get\_data\_size(int argc, char \*argv[], int rank, int numtasks)

{

string input = "";

int size;

// ERROR if too many command line arguments

if (argc > 2) {

if (rank == 0)

cout << "usage: mpirun -np [num MPI tasks] add\_num\_MPI [data size]" << endl;

MPI\_Finalize(); // Terminate MPI

exit(1); // Exit Program

}

// One Command Line Argument Case:

// case where user did not enter number of numbers on command line

// In this case, only one of the MPI processes needs to communicate

// directly with the user. Since there will always be a MPI process

// with rank 0 this is the one that will perform the communication.

if (argc == 1) {

if (rank == 0) {

while (1) {

cout << "Enter the number of numbers to be added:" << endl;

getline(cin, input);

stringstream myStream(input);

if (myStream >> size) break;

cout << "Invalid Input" << endl << endl;

}

}

// since only the root MPI process is communicating with the

// user, the root process must send its value to all of the

// other MPI process. It can do this with the broadcast\_int()

// broadcast routine.

broadcast\_int(&size, 0, rank, numtasks);

}

// Two Command Line Argument case:

// user supplied the number of numbers on the command line.

// Each MPI process can retrieve it from there. No need to

// broadcast it to the other process because each have it at

// run time.

else {

size = atoi(argv[1]);

}

return size;

}

/\*

Routine that fills the number matrix with Random Data with values

between MIN\_VALUE and MAX\_VALUE

This simulates in some way what might happen if there was a

single sequential data acquisition source such as a single file

\*/

void fill\_matrix(double \*numbers, int data\_size)

{

int i;

srand48(SEED);

for (i = 0; i < data\_size; i++) {

numbers[i] = drand48()\*(MAX\_VALUE - MIN\_VALUE) + MIN\_VALUE;

//to verify may want to initialize the numbers array with a pattern

//that has a known answer such as the sum of numbers from 0 to N-1

// The result of that summation is (N+1)\*N/2!!

// numbers[i]=i; // to do so uncomment this line

}

}

/\*

Routine that outputs the numbers matrix to the screen

\*/

void print\_matrix(double \*numbers, int data\_size)

{

int i;

for (i = 0; i < data\_size; i++) {

cout << numbers[i] << endl;

}

}

/\* ONE-TO-ALL SCATTER ROUTINE

Routine to divide and scatter the number data array that resides on the

root MPI process to all other MPI processes in the system.

The number data size is given by the'num\_size' parameter its source

address is given by the '\*numbers' parameter, and the destination

group data associated with the current process is given by the

'\*group' parameter. \*/

void scatter(double \*numbers, double \*group, int num\_size, int root, int rank, int numtasks)

{

MPI\_Status status;

int type = 234;

// determine number of elements in subarray groups to be processed by

// each MPI process assuming a perfectly even distribution of elements

int number\_elements\_per\_section = num\_size / numtasks;

// if root MPI process send portion of numbers array to each of the

// the other MPI processes as well as make a copy of the portion

// of the numbers array that is slated for the root MPI process

if (rank == root) {

int begin\_element = 0;

for (int mpitask = 0; mpitask < numtasks; mpitask++) {

// in MPI root process case just copy the appropriate subsection

// locally from the numbers array over to the group array

if (mpitask == root) {

for (int i = 0; i < number\_elements\_per\_section; i++)

group[i] = numbers[i + begin\_element];

}

// if not the root process send the subsection data to

// the next MPI process

else {

MPI\_Send(&numbers[begin\_element], number\_elements\_per\_section,

MPI\_DOUBLE, mpitask, type, MPI\_COMM\_WORLD);

}

// point to next unsent or uncopied data in numbers array

begin\_element += number\_elements\_per\_section;

}

}

// if a non root process just receive the data

else {

MPI\_Recv(group, number\_elements\_per\_section, MPI\_DOUBLE,

root, type, MPI\_COMM\_WORLD, &status);

}

}

/\*

ALL-TO-ONE Reduce ROUTINE

Routine to accumulate the result of the local summation associated

with each MPI process. This routine takes these partial sums and

produces a global sum on the root MPI process (0)

Input arguments to routine include variable name of local partial

sum of each MPI process. The function returns to MPI root process 0,

the global sum (summation of all partial sums).

\*/

void reduce(double \*sum, double \*partial\_sum, int root, int rank, int numtasks)

{

MPI\_Status status;

int type = 123;

// if MPI root process sum up results from the other p-1 processes

if (rank == root) {

\*sum = \*partial\_sum;

for (int mpitask = 0; mpitask < numtasks; mpitask++) {

if (mpitask != root) {

MPI\_Recv(partial\_sum, 1, MPI\_DOUBLE,

mpitask, type, MPI\_COMM\_WORLD, &status);

(\*sum) += (\*partial\_sum);

}

}

}

// if not root MPI root process then send partial sum to the root

else {

MPI\_Send(partial\_sum, 1, MPI\_DOUBLE,

root, type, MPI\_COMM\_WORLD);

}

}

void reduceMin(double \*min, int root, int rank, int numtasks)

{

MPI\_Status status;

int type = 123;

double localMin = MAX\_VALUE;

// if MPI root process grab minimums from the other p-1 processes

if (rank == root) {

for (int mpitask = 0; mpitask < numtasks; mpitask++) {

if (mpitask != root) {

MPI\_Recv(&localMin, 1, MPI\_DOUBLE,

mpitask, type, MPI\_COMM\_WORLD, &status);

if (localMin < \*min) \*min = localMin;

}

}

}

// if not root MPI root process then send min to the root

else {

MPI\_Send(min, 1, MPI\_DOUBLE,

root, type, MPI\_COMM\_WORLD);

}

}

void reduceMax(double \*max, int root, int rank, int numtasks)

{

MPI\_Status status;

int type = 123;

double localMax = MIN\_VALUE;

// if MPI root process grab the maximums from the other p-1 processes

if (rank == root) {

for (int mpitask = 0; mpitask < numtasks; mpitask++) {

if (mpitask != root) {

MPI\_Recv(&localMax, 1, MPI\_DOUBLE,

mpitask, type, MPI\_COMM\_WORLD, &status);

if (localMax > \*max) \*max = localMax;

}

}

}

// if not root MPI root process then send max to the root

else {

MPI\_Send(max, 1, MPI\_DOUBLE,

root, type, MPI\_COMM\_WORLD);

}

}

/\*

MAIN ROUTINE: summation of numbers in a list

\*/

int main(int argc, char \*argv[])

{

double \*numbers, \*group;

double sum, pt\_sum, min, max;

int data\_size, group\_size, num\_group, i;

int numtasks, rank, num;

MPI\_Status status;

// Initialize a value for the numbers pointer

// Should be able to remove this on dmc, visual studio just throws a fit about

// uninitialized pointer variables.

//double meaningOfLife = 42;

//numbers = &meaningOfLife;

MPI\_Init(&argc, &argv); // initalize MPI environment

MPI\_Comm\_size(MPI\_COMM\_WORLD, &numtasks); // get total number of MPI processes

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); // get unique task id number

//get data size from command line or prompt

//the user for input

data\_size = get\_data\_size(argc, argv, rank, numtasks);

// if root MPI Process (0) then

if (rank == 0) {

// dynamically allocate from heap the numbers array on the root process

numbers = new (nothrow) double[data\_size];

if (numbers == 0) { // check for null pointer

cout << "Memory Allocation Error on Root for numbers array"

<< endl << flush;

MPI\_Abort(MPI\_COMM\_WORLD, 1); // abort the MPI Environment

}

// initialize numbers matrix with random data

fill\_matrix(numbers, data\_size);

// and print the numbers matrix

/\*cout << "numbers matrix =" << endl;

print\_matrix(numbers, data\_size);

cout << endl;\*/

}

// dynamically allocate from heap the group array that will hold

// the partial set of numbers for each MPI process

group = new (nothrow) double[data\_size / numtasks + 1];

if (group == 0) { // check for null pointer to group

cout << "Memory Allocation Error" << endl << flush;

MPI\_Abort(MPI\_COMM\_WORLD, 1); // abort the MPI Environment

}

// scatter the numbers matrix to all processing elements in

// the system

scatter(numbers, group, data\_size, 0, rank, numtasks);

// sum up elements in the group associated with the

// current process

num\_group = data\_size / numtasks; // determine local list size

// group

pt\_sum = 0; // clear out partial sum

min = 0; // initialize min

max = 0; // initialize max

for (i = 0; i < num\_group; i++) {

pt\_sum += group[i];

if (group[i] < min) min = group[i]; // Find the minimum of the group

if (group[i] > max) max = group[i]; // Find the maximum of the group

}

// obtain final sum by summing up partial sums from other MPI tasks

// obtain a global minimum by comparing local minimums from other MPI tasks

// obtain a global maximum by comparing local maximums from other MPI tasks

reduce(&sum, &pt\_sum, 0, rank, numtasks);

reduceMin(&min, 0, rank, numtasks);

reduceMax(&max, 0, rank, numtasks);

// output sum from root MPI process

if (rank == 0) {

cout << "Sum of numbers is " << setprecision(8) << sum << endl;

cout << "Minimum of numbers is " << setprecision(8) << min << endl;

cout << "Maximum of numbers is " << setprecision(8) << max << endl;

}

// reclaim dynamiclly allocated memory

if (rank == 0) delete numbers;

delete group;

// Terminate MPI Program -- perform necessary MPI housekeeping

// clear out all buffers, remove handlers, etc.

MPI\_Finalize();

}

## Output

I don’t think that it was required for us to show output for this revision but I have included it in this report just in case.

uahcls01@dmcvlogin1:Hw2> mpiexec -np 2 add\_num\_MPI\_rev1 2483

Sum of numbers is 64449.072

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 3 add\_num\_MPI\_rev1 2483

Sum of numbers is 64429.463

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 4 add\_num\_MPI\_rev1 2483

Sum of numbers is 64373.874

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 5 add\_num\_MPI\_rev1 2483

Sum of numbers is 64373.874

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 6 add\_num\_MPI\_rev1 2483

Sum of numbers is 64369.325

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 7 add\_num\_MPI\_rev1 2483

Sum of numbers is 64369.325

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 8 add\_num\_MPI\_rev1 2483

Sum of numbers is 64373.874

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 9 add\_num\_MPI\_rev1 2483

Sum of numbers is 64453.198

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 10 add\_num\_MPI\_rev1 2483

Sum of numbers is 64373.874

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 11 add\_num\_MPI\_rev1 2483

Sum of numbers is 64453.198

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 12 add\_num\_MPI\_rev1 2483

Sum of numbers is 64548.838

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 13 add\_num\_MPI\_rev1 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 14 add\_num\_MPI\_rev1 2483

Sum of numbers is 64369.325

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 15 add\_num\_MPI\_rev1 2483

Sum of numbers is 64453.198

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 16 add\_num\_MPI\_rev1 2483

Sum of numbers is 64373.874

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

# Add\_Num\_MPI\_rev2

It should be noted that the implementation for the scatter operation, as well as the grouping afterwards, between revision 2 and revision 3 are different. I had finished revision 2 using an algorithm to scatter the list such that a number from the list is put into an MPI task sequentially until there are no more in the list.

Example :

List = [1, 2, 3, 4, 5, 6]

Number of processes = 4

MPI 1 receives [1, 5]

MPI 2 receives [2, 6]

MPI 3 receives [3]

MPI 4 receives [4]

This didn’t work to well when trying to use the MPI\_Scatter function call in revision 3. After trial and error with the above implementation, when scattering and regrouping, I decided to follow the logic that you provided in class.

## Source Code

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Summation of a Sequence of Numbers Program -- MPI version \*/

/\* September 2017 -- B. Earl Wells -- University of Alabama \*/

/\* in Huntsville \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

This program illustrates the basic concepts of SPMD programming using

MPI. The program represents a common example that is used often in

the CPE 412/512 text, the distributed addition of a sequence of numbers.

The program is written in such a way that it is assumed that the

sequence of numbers is first read from a central source (in this case

a data file) by a single MPI process (the root process) and then

partitioned into equal groups with each part being distributed

(scattered) to the local memory domains of the other MPI

processes in the system. After which each process computes its own

partial sum of the data that is in its domain and then sends this

value to the memory domain of the root process which then

adds the partial sums together (reduce operation) and then

outputs this sum to the screen.

The following is a simplified version of the program which you will

be asked to augment in a number of ways in future homework

assignments. Use the dmc.asc.edu or the Jetson Cluster system for

these assignments.

Notes: This implementation utilizes a minimum set of MPI function

call that include MPI\_Init, MPI\_Finalize, MPI\_Comm\_size,

MPI\_Comm\_rank, MPI\_Send, and MPI\_Recv.

MPI\_Abort is also used to illustrate its functionality.

To compile type:

module load openmpi

mpic++ add\_num\_MPI.cpp -o add\_num\_MPI

To execute:

mpiexec -np [num MPI process] add\_num\_MPI [num of numbers]

EDIT: Kyle Ray

CPE\_512 Intro to Parallel Programming

Homework #2

September 21, 2017

add\_num\_mpi\_rev2.cpp

Addition: Making it so that the application can accept and use

a set of numbers that doesn't have to be a multiple of the number

of processors tasked to do the job.

\*/

using namespace std;

#include <iostream>

#include <iomanip>

#include <sstream>

#include <stdlib.h>

#include <mpi.h> /\* MPI Prototype Header Files \*/

// Defines so that I can compile the code in visual studio

//#define srand48(s) srand(s)

//#define drand48() (((double)rand())/((double)RAND\_MAX))

#define SEED 2397 /\* random number seed \*/

#define MAX\_VALUE 100.0 /\* maximum value of any number in list \*/

#define MIN\_VALUE -50.0 /\* minimum value of any number in list \*/

/\*

ONE-TO-ALL BROADCAST COMMUNICATION ROUTINE

Routine to transfer from the root MPI process the value of

the 'int\_num' parameter to all other MPI processes in the system.

\*/

void broadcast\_int(int \*int\_num, int root, int rank, int numtasks) {

MPI\_Status status;

int type = 123;

// root send value of int\_num to each of the other processes

// using a locally blocking point-to-point send

if (rank == root) {

for (int mpitask = 0; mpitask < numtasks; mpitask++) {

if (mpitask != root) {

MPI\_Send(int\_num, 1, MPI\_INT,

mpitask, type, MPI\_COMM\_WORLD);

}

}

}

// if not root process execute a blocking point-to-point receive

// with the source being to root process and direct this data to

// the local copy of 'int\_num'

else {

MPI\_Recv(int\_num, 1, MPI\_INT,

root, type, MPI\_COMM\_WORLD, &status);

}

}

/\*

Routine to retrieve the data size of the numbers array from the

command line or get this number by prompting the user for the

information. Note: command line values are sent to ALL MPI processes

by the MPI environment.

\*/

int get\_data\_size(int argc, char \*argv[], int rank, int numtasks)

{

string input = "";

int size;

// ERROR if too many command line arguments

if (argc > 2) {

if (rank == 0)

cout << "usage: mpirun -np [num MPI tasks] add\_num\_MPI [data size]" << endl;

MPI\_Finalize(); // Terminate MPI

exit(1); // Exit Program

}

// One Command Line Argument Case:

// case where user did not enter number of numbers on command line

// In this case, only one of the MPI processes needs to communicate

// directly with the user. Since there will always be a MPI process

// with rank 0 this is the one that will perform the communication.

if (argc == 1) {

if (rank == 0) {

while (1) {

cout << "Enter the number of numbers to be added:" << endl;

getline(cin, input);

stringstream myStream(input);

if (myStream >> size) break;

cout << "Invalid Input" << endl << endl;

}

}

// since only the root MPI process is communicating with the

// user, the root process must send its value to all of the

// other MPI process. It can do this with the broadcast\_int()

// broadcast routine.

broadcast\_int(&size, 0, rank, numtasks);

}

// Two Command Line Argument case:

// user supplied the number of numbers on the command line.

// Each MPI process can retrieve it from there. No need to

// broadcast it to the other process because each have it at

// run time.

else {

size = atoi(argv[1]);

}

return size;

}

/\*

Routine that fills the number matrix with Random Data with values

between MIN\_VALUE and MAX\_VALUE

This simulates in some way what might happen if there was a

single sequential data acquisition source such as a single file

\*/

void fill\_matrix(double \*numbers, int data\_size)

{

int i;

srand48(SEED);

for (i = 0; i < data\_size; i++) {

numbers[i] = drand48()\*(MAX\_VALUE - MIN\_VALUE) + MIN\_VALUE;

//to verify may want to initialize the numbers array with a pattern

//that has a known answer such as the sum of numbers from 0 to N-1

// The result of that summation is (N+1)\*N/2!!

// numbers[i]=i; // to do so uncomment this line

}

}

/\*

Routine that outputs the numbers matrix to the screen

\*/

void print\_matrix(double \*numbers, int data\_size)

{

int i;

for (i = 0; i < data\_size; i++) {

cout << numbers[i] << endl;

}

}

/\* ONE-TO-ALL SCATTER ROUTINE

Routine to divide and scatter the number data array that resides on the

root MPI process to all other MPI processes in the system.

The number data size is given by the'num\_size' parameter its source

address is given by the '\*numbers' parameter, and the destination

group data associated with the current process is given by the

'\*group' parameter. \*/

void scatter(double \*numbers, double \*group, int num\_size, int root, int rank, int numtasks)

{

MPI\_Status status;

int type = 234;

// determine number of elements in subarray groups to be processed by

// each MPI process assuming a perfectly even distribution of elements

// krr edits

int base = num\_size / numtasks;

int extra = num\_size % numtasks;

int number\_elements\_per\_section = rank < extra ? base + 2 : base + 1;

// if root MPI process send portion of numbers array to each of the

// the other MPI processes as well as make a copy of the portion

// of the numbers array that is slated for the root MPI process

if (rank == root) {

int begin\_element = 0;

for (int mpitask = 0; mpitask < numtasks; mpitask++) {

// in MPI root process case just copy the appropriate subsection

// locally from the numbers array over to the group array

if (mpitask == root) {

for (int i = 0; i < number\_elements\_per\_section; i++)

group[i] = numbers[i + begin\_element];

}

// if not the root process send the subsection data to

// the next MPI process

else {

MPI\_Send(&numbers[begin\_element], number\_elements\_per\_section,

MPI\_DOUBLE, mpitask, type, MPI\_COMM\_WORLD);

}

// Recalculate number of elements per section

number\_elements\_per\_section = mpitask < extra ? base + 1 : base;

// point to next unsent or uncopied data in numbers array

begin\_element += number\_elements\_per\_section;

}

}

// if a non root process just receive the data

else {

MPI\_Recv(group, number\_elements\_per\_section, MPI\_DOUBLE,

root, type, MPI\_COMM\_WORLD, &status);

}

}

/\*

ALL-TO-ONE Reduce ROUTINE

Routine to accumulate the result of the local summation associated

with each MPI process. This routine takes these partial sums and

produces a global sum on the root MPI process (0)

Input arguments to routine include variable name of local partial

sum of each MPI process. The function returns to MPI root process 0,

the global sum (summation of all partial sums).

\*/

void reduce(double \*sum, double \*partial\_sum, int root, int rank, int numtasks)

{

MPI\_Status status;

int type = 123;

// if MPI root process sum up results from the other p-1 processes

if (rank == root) {

\*sum = \*partial\_sum;

for (int mpitask = 0; mpitask < numtasks; mpitask++) {

if (mpitask != root) {

MPI\_Recv(partial\_sum, 1, MPI\_DOUBLE,

mpitask, type, MPI\_COMM\_WORLD, &status);

(\*sum) += (\*partial\_sum);

}

}

}

// if not root MPI root process then send partial sum to the root

else {

MPI\_Send(partial\_sum, 1, MPI\_DOUBLE,

root, type, MPI\_COMM\_WORLD);

}

}

void reduceMin(double \*min, int root, int rank, int numtasks)

{

MPI\_Status status;

int type = 123;

double localMin = MAX\_VALUE;

// if MPI root process sum up results from the other p-1 processes

if (rank == root) {

for (int mpitask = 0; mpitask < numtasks; mpitask++) {

if (mpitask != root) {

MPI\_Recv(&localMin, 1, MPI\_DOUBLE,

mpitask, type, MPI\_COMM\_WORLD, &status);

if (localMin < \*min) \*min = localMin;

}

}

}

// if not root MPI root process then send partial sum to the root

else {

MPI\_Send(min, 1, MPI\_DOUBLE,

root, type, MPI\_COMM\_WORLD);

}

}

void reduceMax(double \*max, int root, int rank, int numtasks)

{

MPI\_Status status;

int type = 123;

double localMax = MIN\_VALUE;

// if MPI root process sum up results from the other p-1 processes

if (rank == root) {

for (int mpitask = 0; mpitask < numtasks; mpitask++) {

if (mpitask != root) {

MPI\_Recv(&localMax, 1, MPI\_DOUBLE,

mpitask, type, MPI\_COMM\_WORLD, &status);

if (localMax > \*max) \*max = localMax;

}

}

}

// if not root MPI root process then send partial sum to the root

else {

MPI\_Send(max, 1, MPI\_DOUBLE,

root, type, MPI\_COMM\_WORLD);

}

}

/\*

MAIN ROUTINE: summation of numbers in a list

\*/

int main(int argc, char \*argv[])

{

double \*numbers, \*group;

double sum, pt\_sum, min, max;

int data\_size, group\_size, num\_group, i;

int numtasks, rank, num;

MPI\_Status status;

// Initialize a value for the numbers pointer

// Should be able to remove this on dmc, visual studio just throws a fit about

// uninitialized pointer variables.

//double meaningOfLife = 42;

//numbers = &meaningOfLife;

MPI\_Init(&argc, &argv); // initalize MPI environment

MPI\_Comm\_size(MPI\_COMM\_WORLD, &numtasks); // get total number of MPI processes

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); // get unique task id number

//get data size from command line or prompt

//the user for input

data\_size = get\_data\_size(argc, argv, rank, numtasks);

// if root MPI Process (0) then

if (rank == 0) {

// dynamically allocate from heap the numbers array on the root process

numbers = new (nothrow) double[data\_size];

if (numbers == 0) { // check for null pointer

cout << "Memory Allocation Error on Root for numbers array"

<< endl << flush;

MPI\_Abort(MPI\_COMM\_WORLD, 1); // abort the MPI Environment

}

// initialize numbers matrix with random data

fill\_matrix(numbers, data\_size);

// and print the numbers matrix

//cout << "numbers matrix =" << endl;

//print\_matrix(numbers, data\_size);

//cout << endl;

}

// krr going to have to edit this allocation so that the right number

// is allocated for each MPI task

// Pseudo code

int base = data\_size / numtasks;

int extra = data\_size % numtasks;

// dynamically allocate from heap the group array that will hold

// the partial set of numbers for each MPI process

group = rank < extra ? new (nothrow) double[base + 2] : new (nothrow) double[base + 1];

if (group == 0) { // check for null pointer to group

cout << "Memory Allocation Error" << endl << flush;

MPI\_Abort(MPI\_COMM\_WORLD, 1); // abort the MPI Environment

}

// scatter the numbers matrix to all processing elements in

// the system

scatter(numbers, group, data\_size, 0, rank, numtasks);

// sum up elements in the group associated with the

// current process

num\_group = rank < extra ? base + 1 : base;

pt\_sum = 0; // clear out partial sum

min = 0; // initialize min

max = 0; // initialize max

for (i = 0; i < num\_group; i++) {

pt\_sum += group[i];

if (group[i] < min) min = group[i]; // Find the minimum of the group

if (group[i] > max) max = group[i]; // Find the maximum of the group

}

// obtain final sum by summing up partial sums from other MPI tasks

// obtain a global minimum by comparing local minimums from other MPI tasks

// obtain a global maximum by comparing local maximums from other MPI tasks

reduce(&sum, &pt\_sum, 0, rank, numtasks);

reduceMin(&min, 0, rank, numtasks);

reduceMax(&max, 0, rank, numtasks);

// output sum from root MPI process

if (rank == 0) {

cout << "Sum of numbers is " << setprecision(8) << sum << endl;

cout << "Minimum of numbers is " << setprecision(8) << min << endl;

cout << "Maximum of numbers is " << setprecision(8) << max << endl;

}

// reclaim dynamiclly allocated memory

if (rank == 0) delete numbers;

delete group;

// Terminate MPI Program -- perform necessary MPI housekeeping

// clear out all buffers, remove handlers, etc.

MPI\_Finalize();

}

## Output

uahcls01@dmcvlogin1:Hw2> mpiexec -np 2 add\_num\_MPI\_rev2 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 3 add\_num\_MPI\_rev2 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 4 add\_num\_MPI\_rev2 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 5 add\_num\_MPI\_rev2 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 6 add\_num\_MPI\_rev2 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 7 add\_num\_MPI\_rev2 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 8 add\_num\_MPI\_rev2 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 9 add\_num\_MPI\_rev2 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 10 add\_num\_MPI\_rev2 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 11 add\_num\_MPI\_rev2 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 12 add\_num\_MPI\_rev2 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 13 add\_num\_MPI\_rev2 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 14 add\_num\_MPI\_rev2 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 15 add\_num\_MPI\_rev2 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 16 add\_num\_MPI\_rev2 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

## Answers

1. Was the number that was returned for the sum always the same?
   1. Yes, the answer returned from the sum in the revision was always the same, which makes sense because the generated list is the same 2483 numbers each time and we are only varying the number of MPI processes performing the work on this list.

# Add\_Num\_MPI\_rev3

## Source Code

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Summation of a Sequence of Numbers Program -- MPI version \*/

/\* September 2017 -- B. Earl Wells -- University of Alabama \*/

/\* in Huntsville \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

This program illustrates the basic concepts of SPMD programming using

MPI. The program represents a common example that is used often in

the CPE 412/512 text, the distributed addition of a sequence of numbers.

The program is written in such a way that it is assumed that the

sequence of numbers is first read from a central source (in this case

a data file) by a single MPI process (the root process) and then

partitioned into equal groups with each part being distributed

(scattered) to the local memory domains of the other MPI

processes in the system. After which each process computes its own

partial sum of the data that is in its domain and then sends this

value to the memory domain of the root process which then

adds the partial sums together (reduce operation) and then

outputs this sum to the screen.

The following is a simplified version of the program which you will

be asked to augment in a number of ways in future homework

assignments. Use the dmc.asc.edu or the Jetson Cluster system for

these assignments.

Notes: This implementation utilizes a minimum set of MPI function

call that include MPI\_Init, MPI\_Finalize, MPI\_Comm\_size,

MPI\_Comm\_rank, MPI\_Send, and MPI\_Recv.

MPI\_Abort is also used to illustrate its functionality.

To compile type:

module load openmpi

mpic++ add\_num\_MPI.cpp -o add\_num\_MPI

To execute:

mpiexec -np [num MPI process] add\_num\_MPI [num of numbers]

EDIT: Kyle Ray

CPE\_512 Intro to Parallel Programming

Homework #2

September 21, 2017

add\_num\_mpi\_rev3.cpp

Addition: Replacing the broadcast, scatter, and reduce calls with the

appropriate MPI built in call.

\*/

using namespace std;

#include <iostream>

#include <iomanip>

#include <sstream>

#include <stdlib.h>

#include <mpi.h> /\* MPI Prototype Header Files \*/

#include <cmath> // ceil

// Defines so that I can compile the code in visual studio

//#define srand48(s) srand(s)

//#define drand48() (((double)rand())/((double)RAND\_MAX))

#define SEED 2397 /\* random number seed \*/

#define MAX\_VALUE 100.0 /\* maximum value of any number in list \*/

#define MIN\_VALUE -50.0 /\* minimum value of any number in list \*/

/\*

ONE-TO-ALL BROADCAST COMMUNICATION ROUTINE

Routine to transfer from the root MPI process the value of

the 'int\_num' parameter to all other MPI processes in the system.

\*/

void broadcast\_int(int \*int\_num, int root, int rank, int numtasks) {

MPI\_Status status;

int type = 123;

// root send value of int\_num to each of the other processes

// using a locally blocking point-to-point send

if (rank == root) {

for (int mpitask = 0; mpitask < numtasks; mpitask++) {

if (mpitask != root) {

MPI\_Send(int\_num, 1, MPI\_INT,

mpitask, type, MPI\_COMM\_WORLD);

}

}

}

// if not root process execute a blocking point-to-point receive

// with the source being to root process and direct this data to

// the local copy of 'int\_num'

else {

MPI\_Recv(int\_num, 1, MPI\_INT,

root, type, MPI\_COMM\_WORLD, &status);

}

}

/\*

Routine to retrieve the data size of the numbers array from the

command line or get this number by prompting the user for the

information. Note: command line values are sent to ALL MPI processes

by the MPI environment.

\*/

int get\_data\_size(int argc, char \*argv[], int rank, int numtasks)

{

string input = "";

int size;

// ERROR if too many command line arguments

if (argc > 2) {

if (rank == 0)

cout << "usage: mpirun -np [num MPI tasks] add\_num\_MPI [data size]" << endl;

MPI\_Finalize(); // Terminate MPI

exit(1); // Exit Program

}

// One Command Line Argument Case:

// case where user did not enter number of numbers on command line

// In this case, only one of the MPI processes needs to communicate

// directly with the user. Since there will always be a MPI process

// with rank 0 this is the one that will perform the communication.

if (argc == 1) {

if (rank == 0) {

while (1) {

cout << "Enter the number of numbers to be added:" << endl;

getline(cin, input);

stringstream myStream(input);

if (myStream >> size) break;

cout << "Invalid Input" << endl << endl;

}

}

// since only the root MPI process is communicating with the

// user, the root process must send its value to all of the

// other MPI process. It can do this with the broadcast\_int()

// broadcast routine.

//broadcast\_int(&size, 0, rank, numtasks);

MPI\_Bcast(&size, numtasks, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

}

// Two Command Line Argument case:

// user supplied the number of numbers on the command line.

// Each MPI process can retrieve it from there. No need to

// broadcast it to the other process because each have it at

// run time.

else {

size = atoi(argv[1]);

}

return size;

}

/\*

Routine that fills the number matrix with Random Data with values

between MIN\_VALUE and MAX\_VALUE

This simulates in some way what might happen if there was a

single sequential data acquisition source such as a single file

\*/

void fill\_matrix(double \*numbers, int data\_size)

{

int i;

srand48(SEED);

for (i = 0; i < data\_size; i++) {

numbers[i] = drand48()\*(MAX\_VALUE - MIN\_VALUE) + MIN\_VALUE;

//to verify may want to initialize the numbers array with a pattern

//that has a known answer such as the sum of numbers from 0 to N-1

// The result of that summation is (N+1)\*N/2!!

// numbers[i]=i; // to do so uncomment this line

}

}

/\*

Routine that outputs the numbers matrix to the screen

\*/

void print\_matrix(double \*numbers, int data\_size)

{

int i;

for (i = 0; i < data\_size; i++) {

cout << numbers[i] << endl;

}

}

/\* ONE-TO-ALL SCATTER ROUTINE

Routine to divide and scatter the number data array that resides on the

root MPI process to all other MPI processes in the system.

The number data size is given by the'num\_size' parameter its source

address is given by the '\*numbers' parameter, and the destination

group data associated with the current process is given by the

'\*group' parameter. \*/

void scatter(double \*numbers, double \*group, int num\_size, int root, int rank, int numtasks)

{

MPI\_Status status;

int type = 234;

// determine number of elements in subarray groups to be processed by

// each MPI process assuming a perfectly even distribution of elements

// krr edits

int number\_elements\_per\_section = ceil((double)num\_size / numtasks);

// if root MPI process send portion of numbers array to each of the

// the other MPI processes as well as make a copy of the portion

// of the numbers array that is slated for the root MPI process

if (rank == root) {

int begin\_element = 0;

for (int mpitask = 0; mpitask < numtasks; mpitask++) {

// in MPI root process case just copy the appropriate subsection

// locally from the numbers array over to the group array

if (mpitask == root) {

for (int i = 0; i < number\_elements\_per\_section; i++)

group[i] = numbers[i + begin\_element];

}

// if not the root process send the subsection data to

// the next MPI process

else {

MPI\_Send(&numbers[begin\_element], number\_elements\_per\_section,

MPI\_DOUBLE, mpitask, type, MPI\_COMM\_WORLD);

}

// point to next unsent or uncopied data in numbers array

begin\_element += number\_elements\_per\_section;

}

}

// if a non root process just receive the data

else {

MPI\_Recv(group, number\_elements\_per\_section, MPI\_DOUBLE,

root, type, MPI\_COMM\_WORLD, &status);

}

}

/\*

ALL-TO-ONE Reduce ROUTINE

Routine to accumulate the result of the local summation associated

with each MPI process. This routine takes these partial sums and

produces a global sum on the root MPI process (0)

Input arguments to routine include variable name of local partial

sum of each MPI process. The function returns to MPI root process 0,

the global sum (summation of all partial sums).

\*/

void reduce(double \*sum, double \*partial\_sum, int root, int rank, int numtasks)

{

MPI\_Status status;

int type = 123;

// if MPI root process sum up results from the other p-1 processes

if (rank == root) {

\*sum = \*partial\_sum;

for (int mpitask = 0; mpitask < numtasks; mpitask++) {

if (mpitask != root) {

MPI\_Recv(partial\_sum, 1, MPI\_DOUBLE,

mpitask, type, MPI\_COMM\_WORLD, &status);

(\*sum) += (\*partial\_sum);

}

}

}

// if not root MPI root process then send partial sum to the root

else {

MPI\_Send(partial\_sum, 1, MPI\_DOUBLE,

root, type, MPI\_COMM\_WORLD);

}

}

void reduceMin(double \*min, int root, int rank, int numtasks)

{

MPI\_Status status;

int type = 123;

double localMin = MAX\_VALUE;

// if MPI root process sum up results from the other p-1 processes

if (rank == root) {

for (int mpitask = 0; mpitask < numtasks; mpitask++) {

if (mpitask != root) {

MPI\_Recv(&localMin, 1, MPI\_DOUBLE,

mpitask, type, MPI\_COMM\_WORLD, &status);

if (localMin < \*min) \*min = localMin;

}

}

}

// if not root MPI root process then send partial sum to the root

else {

MPI\_Send(min, 1, MPI\_DOUBLE,

root, type, MPI\_COMM\_WORLD);

}

}

void reduceMax(double \*max, int root, int rank, int numtasks)

{

MPI\_Status status;

int type = 123;

double localMax = MIN\_VALUE;

// if MPI root process sum up results from the other p-1 processes

if (rank == root) {

for (int mpitask = 0; mpitask < numtasks; mpitask++) {

if (mpitask != root) {

MPI\_Recv(&localMax, 1, MPI\_DOUBLE,

mpitask, type, MPI\_COMM\_WORLD, &status);

if (localMax > \*max) \*max = localMax;

}

}

}

// if not root MPI root process then send partial sum to the root

else {

MPI\_Send(max, 1, MPI\_DOUBLE,

root, type, MPI\_COMM\_WORLD);

}

}

/\*

MAIN ROUTINE: summation of numbers in a list

\*/

int main(int argc, char \*argv[])

{

double \*numbers, \*group;

double sum, pt\_sum, min, max;

int data\_size, group\_size, num\_group, i;

int numtasks, rank, num;

MPI\_Status status;

// krr edits

int\* scounts, \*displs;

int displs\_idx, base, extra;

// Initialize a value for the numbers pointer

// Should be able to remove this on dmc, visual studio just throws a fit about

// uninitialized pointer variables.

//double meaningOfLife = 42;

//numbers = &meaningOfLife;

MPI\_Init(&argc, &argv); // initalize MPI environment

MPI\_Comm\_size(MPI\_COMM\_WORLD, &numtasks); // get total number of MPI processes

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); // get unique task id number

//get data size from command line or prompt

//the user for input

data\_size = get\_data\_size(argc, argv, rank, numtasks);

// if root MPI Process (0) then

if (rank == 0) {

// dynamically allocate from heap the numbers array on the root process

numbers = new (nothrow) double[data\_size];

if (numbers == 0) { // check for null pointer

cout << "Memory Allocation Error on Root for numbers array"

<< endl << flush;

MPI\_Abort(MPI\_COMM\_WORLD, 1); // abort the MPI Environment

}

// initialize numbers matrix with random data

fill\_matrix(numbers, data\_size);

// and print the numbers matrix

/\*cout << "numbers matrix =" << endl;

print\_matrix(numbers, data\_size);

cout << endl;\*/

}

// Base line number of tasks each MPI process will have

base = ceil((double)data\_size / numtasks);

// dynamically allocate from heap the group array that will hold

// the partial set of numbers for each MPI process

group = new (nothrow) double[base + 1]; // everyone gets the same size

if (group == 0) { // check for null pointer to group

cout << "Memory Allocation Error" << endl << flush;

MPI\_Abort(MPI\_COMM\_WORLD, 1); // abort the MPI Environment

}

scounts = new (nothrow) int[numtasks];

displs = new (nothrow) int[numtasks];

displs\_idx = 0;

// Get the counts and displacements

for (int mpitask = 0; mpitask < numtasks; mpitask++)

{

if (base\*(rank + 1) <= data\_size)

scounts[mpitask] = base;

else if (base \* rank < data\_size)

scounts[mpitask] = data\_size - base\*rank;

else

scounts[mpitask] = 0;

displs[mpitask] = displs\_idx;

displs\_idx += scounts[mpitask];

}

// scatter the numbers matrix to all processing elements in

// the system

//scatter(numbers, group, data\_size, 0, rank, numtasks);

// KRR TEST MPI\_Scatterv

//MPI\_Scatter(numbers, data\_size, MPI\_DOUBLE, group, base + 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Scatterv(numbers, scounts, displs, MPI\_DOUBLE, group, base + 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

// Calculate the number in the group distribution

if (base\*(rank + 1) <= data\_size)

num\_group = base;

else if (base \* rank < data\_size)

num\_group = data\_size - base\*rank;

else

num\_group = 0;

// sum up elements in the group associated with the

// current process

pt\_sum = 0; // clear out partial sum

min = 0; // initialize min

max = 0; // initialize max

for (i = 0; i < num\_group; i++) {

pt\_sum += group[i];

if (group[i] < min) min = group[i]; // Find the minimum of the group

if (group[i] > max) max = group[i]; // Find the maximum of the group

}

// obtain final sum by summing up partial sums from other MPI tasks

// obtain a global minimum by comparing local minimums from other MPI tasks

// obtain a global maximum by comparing local maximums from other MPI tasks

// edit to use the MPI reduce

double min\_final;

double max\_final;

MPI\_Reduce(&pt\_sum, &sum, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

MPI\_Reduce(&min, &min\_final, 1, MPI\_DOUBLE, MPI\_MIN, 0, MPI\_COMM\_WORLD);

MPI\_Reduce(&max, &max\_final, 1, MPI\_DOUBLE, MPI\_MAX, 0, MPI\_COMM\_WORLD);

/\*reduce(&sum, &pt\_sum, 0, rank, numtasks);

reduceMin(&min, 0, rank, numtasks);

reduceMax(&max, 0, rank, numtasks);\*/

// output sum from root MPI process

if (rank == 0) {

cout << "Sum of numbers is " << setprecision(8) << sum << endl;

cout << "Minimum of numbers is " << setprecision(8) << min\_final << endl;

cout << "Maximum of numbers is " << setprecision(8) << max\_final << endl;

}

// reclaim dynamiclly allocated memory

if (rank == 0) delete numbers;

delete group;

delete scounts;

delete displs;

// Terminate MPI Program -- perform necessary MPI housekeeping

// clear out all buffers, remove handlers, etc.

MPI\_Finalize();

}

## Output

uahcls01@dmcvlogin1:Hw2> mpiexec -np 2 add\_num\_MPI\_rev3 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 3 add\_num\_MPI\_rev3 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 4 add\_num\_MPI\_rev3 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 5 add\_num\_MPI\_rev3 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 6 add\_num\_MPI\_rev3 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 7 add\_num\_MPI\_rev3 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 8 add\_num\_MPI\_rev3 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 9 add\_num\_MPI\_rev3 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 10 add\_num\_MPI\_rev3 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 11 add\_num\_MPI\_rev3 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 12 add\_num\_MPI\_rev3 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 13 add\_num\_MPI\_rev3 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 14 add\_num\_MPI\_rev3 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 15 add\_num\_MPI\_rev3 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

uahcls01@dmcvlogin1:Hw2> mpiexec -np 16 add\_num\_MPI\_rev3 2483

Sum of numbers is 64427.527

Minimum of numbers is -49.860801

Maximum of numbers is 99.984658

## Answers

1. Was the number that was returned for the sum always the same?
   1. Yes, the answer returned from the sum in the revision was always the same, which makes sense because the generated list is the same 2483 numbers each time and we are only varying the number of MPI processes performing the work on this list.