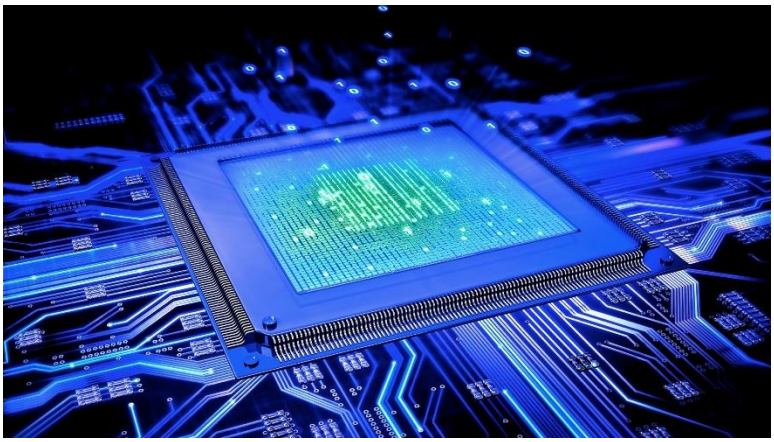

Homework #2
ADD_NUM_MPI

CPE 512

KYLE RAY

August 31, 2017

Contents

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?
    a. 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?
    a. 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.