CSCI 4/5576 HW4 Grading Rubric University of Colorado at Boulder

Last Name:

First Name:

HW4 - Butterfly Allreduce and Performance Analysis

Rubric	Maximum Points	Points Received
Tree-based “butterfly” Allreduce Diagrams	10	
Tree-based “butterfly” Allreduce Routines	20	
Instrumentation and Timing	30	
Plots and Performance Analysis	30	
Report and Discussion Questions	10	

Total: / 100

Please review the report for comments and grading feedback.

Additional grader remarks:

```c
/*
 * Brandon Bell
 * csci4576
 * hw4 9-14-2016
 *
 * Homework 4: Butterfly AllReduce and Performance Analysis.
*/

#include <stdio.h>
#include <string.h>
#include "mpi.h"

int MyPI_Bcast (
                 void*          message,
                 int*           count,
                 MPI_Datatype   datatype,
                 int            root,
                 MPI_Comm       comm,
                 int            bitOrder
                )
{
    // Determine the number of stages required for the tree
    // I'm nabbing the Ceiling_log2() algorythim form chapter 5 in the book to
    // handle this.
    int p;
    int my_rank;
    MPI_Status  status;
    MPI_Comm_size ( comm, &p );
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    unsigned int stages      = 0;
    unsigned int temp   = (unsigned) p - 1;
    unsigned int companion;

    while ( temp != 0 )
    {
        temp   = temp >> 1;
        stages = stages + 1;
    }

    // Re-map Mpi rank into a rankspace where rank 0 is the root. The broadcast
    // algorythm is based an bit arithmatic for a root with value zero and this
    // process zero must have the data beinf broadcast so, re-map the mpi rank
    // to root = rank zero with modulo arithmatic, the ultimate mapping is
    // irrelevant ( I think) as long as it is consitent and the new rank zero
    // has the messsage to be sent.

    int rank = ( my_rank + ( p - root ) ) % p;
    int mpirank;// = ( rank + ( p + root ) ) % p;
    int mpic;

    // Set up the loop to determine send and recvs for each stage of the tree.
    // This is for low to high bit method.
    for ( int stage = 0; stage < stages; stage++ )
    {
        // Determine the current processes send/Recev companion.
        companion = ( 1 << stage ) ^ rank;

        // Detremine wheither to send/recv or do nothing with it's companion.
        // First determine if p is to recv at this stage.
        if ( ( rank < companion ) && ( companion >> ( 1 << stage ) ) == 0 )
        {
            mpic = ( companion + ( p + root ) ) % p;
            mpirank = ( rank + ( p + root ) ) % p;
            printf(" send mpirank %d mpic %d\n", mpirank, mpic );
            MPI_Send( message, count, MPI_FLOAT, mpirank, 1, comm );
            printf("Step %d rank %d Sends to %d\n", stage, rank, companion);
        }
        else if ( ( rank > companion ) && ( companion >> ( 1 << stage ) ) == 0 )
        {
            mpic = ( companion + ( p + root ) ) % p;
            mpirank = ( rank + ( p + root ) ) % p;
            printf(" recv mpirank %d mpic %d\n", mpirank, mpic );
            MPI_Recv( message, count, MPI_FLOAT, mpirank, 1, comm, &status );
            printf("Step %d rank %d Recvs from %d\n", stage, rank, companion);
        }
        // Or if p is suposed to send at this stage.
        // Handle the case that P and it's companion are not to communicate
        // during this stage.
        else
        {
            mpirank = ( companion + ( p + root ) ) % p;
            mpirank = ( rank + ( p + root ) ) % p;
            printf("Step %d rank %d does nothing\n", stage, mpirank);
            continue;
        }
    }
    return message;

}

void MyPI_Reduce (
                  void*         operand,
                  void*         result,
                  int           count,
                  MPI_Datatype  datatype,
                  MPI_Op        operator,
                  int           root,
                  MPI_Comm      comm,
                  int           bitOrder
                  )
{
}

void MyPI_AllReduce (
                     void*          operand,
                     void*          result,
                     int            count,
                     MPI_Datatype   datatype,
                     MPI_Op         operator,
                     int            root,
                     MPI_Comm       comm,
                     int            bitOrder
                     )
{
}

void MyPI_AllReduce_Trivial (
                             void*          operand,
                             void*          result,
                             int            count,
                             MPI_Datatype   datatype,
                             MPI_Op         operator,
                             int            root,
                             MPI_Comm       comm,
                             int            bitOrder
                             )
```

```
{
    MPI_Status  status;
    MPI_Comm_size ( comm, &p );
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // P0 only recvs.
    if (my_rank == 0)
    {
        source = 1;
        MPI_Recv( &result, count, MPI_FLOAT, source, tag, comm, &status );
        total = total + integral;
    }
    //
    else
    {
        source = 0;
        MPI_Send( &result, count, MPI_FLOAT, dest, tag, MPI_COMM_WORLD );
    }
}

int main ( int argc, char* argv[] )
{
    int         my_rank;
    int         p;
    int         source;
    int         dest;
    int         tag = 0;
    float       message;
    MPI_Status  status;

    // Spin-up Mpi.
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    // Check to make sure that more than process are running.
    if ( p == 1 )
    {
        printf("Please Run with > 1 processes.\n" );
        return 1;
    }

    if ( my_rank == 0 )
        message = 5;
    else
        message = 0;

    //printf("Message on rank %d is %f \n", my_rank, message );

    MyPI_Bcast ( &message, 1, MPI_FLOAT, 0, MPI_COMM_WORLD, 1 );

    printf("Message on rank %d is %f \n", my_rank, message );

    MPI_Finalize();
    return 0;
}
```