```c
/*
 * Brandon Bell
 * csci4576
 * hw2 8-31-2016
 *
 * Part 1: Ring Hello World program.
*/

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

int main( int argc, char* argv[] )
{
    int         my_rank;       /* rank of process      */
    int         p;             /* number of processes  */
    int         source;        /* rank of sender       */
    int         dest;          /* rank of receiver     */
    int         tag = 0;       /* tag for messages     */
    char        message[100];  /* storage for message  */
    MPI_Status  status;        /* return status for    */
                               /* receive              */

    // Initialize MPI and fetch p's rank and comm size.
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &my_rank );
    MPI_Comm_size( MPI_COMM_WORLD, &p );

     /* Setup the ring of process comunication using modulo arithimatic. First
      * check to see if p is alone in the universe and if so, just print a
      * meassage as there's no need to invoke MPI. If not, start a message
      * passing ring with process 0. p0 will send and then recive a message, all
      * other p's recive and then send. */
    if ( p == 1 )
    {
        sprintf( message, "Greetings from process 0" );
        printf( "%s\n", message );
    }
    else if ( my_rank == 0 )
    {
        sprintf( message, "Greetings from process 0" );
        dest   = 1;
        source = p - 1;

        // Send message to p1 and then listen and print message from process p.
        MPI_Send( message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD );
        MPI_Recv( message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status );
        printf("%s\n", message);
    }
    else
    {
        /* dest of new message and source incoming message. The modulo p in
        * dest ensures that if p is last process, it sends it's message to p0. */
        dest   = (my_rank + 1) % p;
        source = my_rank - 1;

        // Receve Message from previous process and print it.
        MPI_Recv( message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status );
        printf("%s\n", message);

        // Craft and send a new message to the next process.
        sprintf( message, "Greetings from process %d", my_rank );
        MPI_Send( message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD );
    }
    // Close-up shop.
    MPI_Finalize();
}
```

```c
/*
 * Brandon Bell
 * csci4576
 * hw2 8-31-2016
 *
 * Part 2: Trapazoid Rule with linear reduce.
 *
 * I modified the Source code from the book to use a linear reduce algorithm
 * rather than it's original all-to-one reduce. Code retrived from,
 * http://www.cs.usfca.edu/~peter/ppmpi/ppmpi_c.tar.z
*/

/* trap.c -- Parallel Trapezoidal Rule, first version
 *
 * Input: None.
 * Output:  Estimate of the integral from a to b of f(x)
 *    using the trapezoidal rule and n trapezoids.
 *
 * Algorithm:
 *    1.  Each process calculates "its" interval of
 *        integration.
 *    2.  Each process estimates the integral of f(x)
 *        over its interval using the trapezoidal rule.
 *    3a. Each process != 0 sends its integral to 0.
 *    3b. Process 0 sums the calculations received from
 *        the individual processes and prints the result.
 *
 * Notes:
 *    1.  f(x), a, b, and n are all hardwired.
 *    2.  The number of processes (p) should evenly divide
 *        the number of trapezoids (n = 1024)
 *
 * See Chap. 4, pp. 56 & ff. in PPMPI.
 */
#include <stdio.h>

/* We'll be using MPI routines, definitions, etc. */
#include "mpi.h"


int main(int argc, char** argv) {
    int         my_rank;   /* My process rank           */
    int         p;         /* The number of processes   */
    float       a = 0.0;   /* Left endpoint             */
    float       b = 1.0;   /* Right endpoint            */
    int         n = 1024;  /* Number of trapezoids      */
    float       h;         /* Trapezoid base length     */
    float       local_a;   /* Left endpoint my process  */
    float       local_b;   /* Right endpoint my process */
    int         local_n;   /* Number of trapezoids for  */
                           /* my calculation            */
    float       integral;  /* Integral over my interval */
    float       total;     /* Total integral            */
    int         source;    /* Process sending integral  */
    int         dest;      /* All messages go to 0      */
    int         tag = 0;
    MPI_Status  status;

    float Trap(float local_a, float local_b, int local_n,
              float h);    /* Calculate local integral  */

    /* Let the system do what it needs to start up MPI */
    MPI_Init(&argc, &argv);

    /* Get my process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out how many processes are being used */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    if ( p == 1 )
    {
        printf("Please Run with > 1 processes." );
        return 1;
    }

    h = (b-a)/n;    /* h is the same for all processes */
    local_n = n/p;  /* So is the number of trapezoids */

    /* Length of each process' interval of
     * integration = local_n*h.  So my interval
     * starts at: */
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    integral = Trap(local_a, local_b, local_n, h);

    /*
     * Sum the individual trapazoids with a linear reduce. Each p sends it's
     * total to p-1 and p0 prints the results. This conditional branch is the
     * only code that I've modified, other than adding an int type to main to
     * shut the compiler up.
     */
    // p0 only receives from p1.
    if (my_rank == 0)
    {
        source = 1;
        MPI_Recv( &total, 1, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &status );
        total = total + integral;
    }
    // process p only sends to the next process.
    else if ( my_rank == ( p - 1 ) )
    {
        dest = my_rank - 1;
        MPI_Send( &integral, 1, MPI_FLOAT, dest, tag, MPI_COMM_WORLD );
    }
    // everybody else in between p and p0 recvs, adds, and sends.
    else
    {
        source = my_rank + 1;
        dest   = my_rank - 1;
        MPI_Recv( &total, 1, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &status );
        // each p adds it's integral to the running total.
        total = total + integral;
        MPI_Send( &total, 1, MPI_FLOAT, dest, tag, MPI_COMM_WORLD );
    }


    /* Print the result */
    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n",
            n);
        printf("of the integral from %f to %f = %f\n",
            a, b, total);
    }

    /* Shut down MPI */
```

```
    MPI_Finalize();
} /*  main  */


float Trap(
          float  local_a   /* in */,
          float  local_b   /* in */,
          int    local_n   /* in */,
          float  h         /* in */) {

    float integral;   /* Store result in integral  */
    float x;
    int i;

    float f(float x); /* function we're integrating */

    integral = (f(local_a) + f(local_b))/2.0;
    x = local_a;
    for (i = 1; i <= local_n-1; i++) {
        x = x + h;
        integral = integral + f(x);
    }
    integral = integral*h;
    return integral;
} /*  Trap  */


float f(float x) {
    float return_val;
    /* Calculate f(x). */
    /* Store calculation in return_val. */
    return_val = x*x;
    return return_val;
} /* f */
```

```c
/*
 * Brandon Bell
 * csci4576
 * hw2 8-31-2016
 *
 * Part 3a: PPMPI 4.7.2-a Simpsons Rule Serial Program.
 */

#include <stdio.h>

// I nabbed f from the trap program.
float f(float x) {
    float return_val;
    /* Calculate f(x). */
    /* Store calculation in return_val. */
    return_val = x*x;
    return return_val;
}

float simpson (
          float  a   /* in */,
          float  b   /* in */,
          int    n   /* in */,
          float  h   /* in */)
{
   float integral;
   float x = a;

   // Take care of the starting and end points.
   integral = f(a) + f(b);
   // Loop through the n strips in the region of integration.
   for ( int i=1; i<(2*n - 1); i++ )
   {
        // increment x by a half interval at a time (we need the mid point).
        x += h / 2;
        if ( i % 2 == 0 )
            integral += ( 2 * f(x) );
        else
            integral += ( 4 * f(x) );
   }
   integral *= ( h / 3 );
   return integral;
}

int main(int argc, char** argv)
{
    float  total;
    float  a = 0;
    float  b = 1;
    int    n = 1024;
    float  h;

    h = (b-a)/n;
    total = simpson( a, b, n, h );
    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %f\n", a, b, total);
}
```

```c
/*
 * Brandon Bell
 * csci4576
 * hw2 8-31-2016
 *
 * Part 3a: PPMPI 4.7.2-a Simpsons Rule Serial Program.
 */

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

// I nabbed f from the trap program.
float f(float x) {
    float return_val;
    /* Calculate f(x). */
    /* Store calculation in return_val. */
    return_val = x*x;
    return return_val;
}

// I nabbed simpson from my serial code.
float simpson (
          float  a   /* in */,
          float  b   /* in */,
          int    n   /* in */,
          float  h   /* in */)
{
   float integral;
   float x = a;

   // Take care of the starting and end points.
   integral = f(a) + f(b);
   // Loop through the n strips in the region of integration.
   for ( int i=1; i<(2*n - 1); i++ )
   {
        // increment x by a half interval at a time (we need the mid point).
        x += h / 2;
        if ( i % 2 == 0 )
            integral += ( 2 * f(x) );
        else
            integral += ( 4 * f(x) );
   }
   integral *= ( h / 3 );
   return integral;
}

// Most of my main is addapted from the Parallel Trap Code.
int main(int argc, char** argv)
{
    int         my_rank;   /* My process rank           */
    int         p;         /* The number of processes   */
    float       a = 0.0;   /* Left endpoint             */
    float       b = 1.0;   /* Right endpoint            */
    int         n = 1024;  /* Number of trapezoids      */
    float       h;         /* Trapezoid base length     */
    float       local_a;   /* Left endpoint my process  */
    float       local_b;   /* Right endpoint my process */
    int         local_n;   /* Number of trapezoids for  */
                           /* my calculation            */
    float       integral;  /* Integral over my interval */
    float       total;     /* Total integral            */
    int         source;    /* Process sending integral  */
    int         dest;      /* All messages go to 0      */
    int         tag = 0;
    int         verbose = 0; // For determining verbosity level.
    MPI_Status  status;

    // Initialize MPI and retreive world size and p's rank.
    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." );
        return 1;
    }


    // Parse the comand line arguments. I'm sure this can be done better,
    // especially in a parallel enviroment but, I'm tierd and want bed:)
    if ( argc > 1 )
    {
        // Loop though all the arguments if more than on is present.
        for ( int a=1; a<argc; a++ )
        {
            // checks for -i and a convertable int value.
            if ( !strcmp(argv[a],"-i") && (argc-1) > a && strtol( argv[a+1], NULL, 10 ) )
            {
                n = strtol( argv[2], NULL, 10 );
                // Ensure n is even and pass the loop over the int value.
                if ( n%2 != 0 )
                    n++;
                else if ( n == 0)
                    n = 1024;
                a++;
            }
            // checks for verbose flags.
            else if ( !strcmp(argv[a],"-v") || !strcmp(argv[a],"--verbose") )
                verbose = 1;
            else
            {
                if (my_rank == 0)
                {
                    printf("==> Invalid arguments\n");
                    printf("Usage: cmd {-i [int],-v,--verbose}\n");
                    break;
                }
            }
        }
    }

    // Dertrimine the interval of integration for n bins. ( Common to all ps ).
    h = (b-a)/n;
    // Determine the number of bins in the interval of each p.
    local_n = n/p;

    /* Length of each process' interval of
     * integration = local_n*h.  So my interval
     * starts at: */
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    integral = simpson(local_a, local_b, local_n, h);
```

```
/*
 * Sum the individual regions with a linear reduce. Each p sends it's
 * total to p-1 and p0 prints the results. Taken from my Paralled Trap
 * code.
 */
// p0 only receives from p1.
if (my_rank == 0)
{
    source = 1;
    MPI_Recv( &total, 1, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &status );
    total = total + integral;
}
// process p only sends to the next process.
else if ( my_rank == ( p - 1 ) )
{
    dest = my_rank - 1;
    MPI_Send( &integral, 1, MPI_FLOAT, dest, tag, MPI_COMM_WORLD );
}
// everybody else in between p and p0 recvs, adds, and sends.
else
{
    source = my_rank + 1;
    dest   = my_rank - 1;
    MPI_Recv( &total, 1, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &status );
    total = total + integral;
    MPI_Send( &total, 1, MPI_FLOAT, dest, tag, MPI_COMM_WORLD );
}

// Concludeing output by Process zero. My concludeing output is I'm at UI.
if ( my_rank == 0 )
{
    // Handle the output of the p's partitions if verbose switch set.
    if ( verbose == 1 )
    {
        // Handle the initial process.
        printf("Process 0: [ 1 ");
        for( int t=1; t<local_n; t++ )
        {
            if ( t%2 == 0 )
                printf("2 ");
            else
                printf("4 ");
        }
        printf("]\n");
        // Handle the intermediat Processes.
        for ( int z=1; z < p-1; z++ )
        {
            printf("Process %d: [ ",z);
            for( int t=0; t<local_n; t++ )
            {
                if ( t%2 == 0 )
                    printf("2 ");
                else
                    printf("4 ");
            }
            printf("]\n");
        }
        // Handle the last process.
        printf("Process %d: [ ",p-1);
        for( int t=0; t<local_n-1; t++ )
        {
            if ( t%2 == 0 )
                printf("2 ");
            else
                printf("4 ");
        }
        printf("1 ]\n");
    }
    printf("With n = %d intervals, our estimate\n", n);
    printf("of the integral from %f to %f = %f\n", a, b, total);
}

/* Shut down MPI */
MPI_Finalize();
}
```