Parallel Programming Program 2 Report

<u>Parallelization strategies:</u>

- How did you divide work among remote hosts?
- I used row partitioning to divide bucket into various blocks which will be assigned to each remote host to compute Schrodinger's wave equation.
    - I have taken Quotient and Reminder of size (number of rows) to total number of remote hosts
    - I wanted to divide bucket work equally as much as possible among all hosts. So, each host gets at least "Quotient" number of rows
    - If there is a remainder, I allocated 1 row to each host until rank becomes equal or greater than the remainder, starting from rank 0 host.

    Example: size = 100, hosts = 6 Quotient = 16, Remainder = 4
    Host 1 = 16 + 1, Host 2 = 16 + 1, Host 3 = 16 + 1, Host 4 = 16 + 1, Host 5 = 16, Host 6 = 16

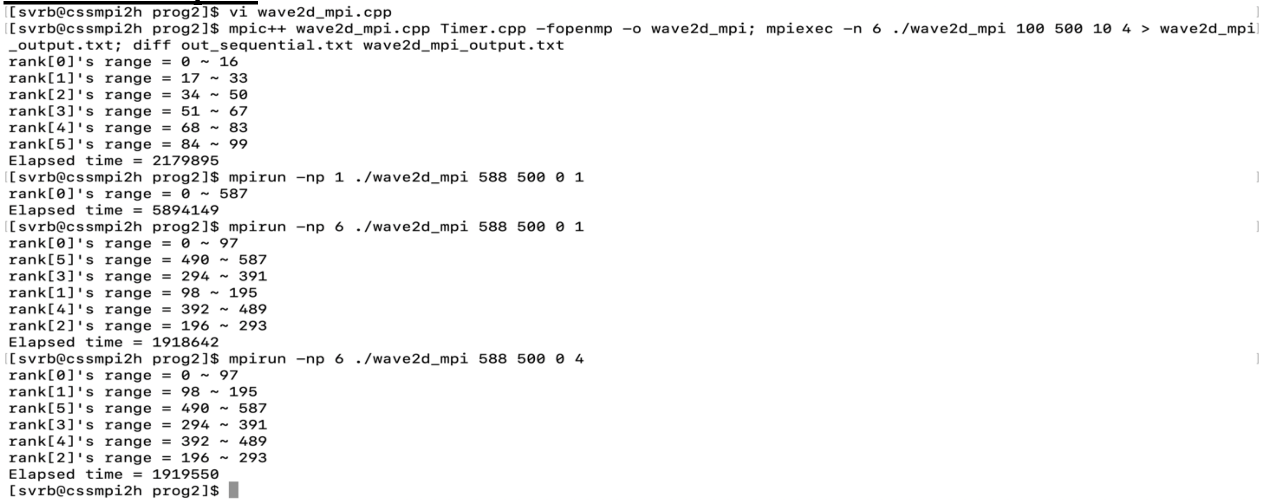
- Calculating heights for all cells of bucket in distributed manner
    - Based on partitioning as described, I calculated IStart and IEnd variables which denote starting row and ending row of every host.
    - With IStart, IEnd and all columns, I wrote two for loops to calculate heights of cells
        - Note: with exception that first row, first column, last row and last column have omitted from for loop
    - There is a requirement that rank = 0 host (master) is responsible to print height values, so each worker host must send entire block of calculated heights to master node
        - "*MPI_Send(&(z[t%3][iStart][0]), size * rowsAllocated, MPI_DOUBLE, 0, my_rank, MPI_COMM_WORLD);*"
        - The above send request will be common to all the workers
            - data - will be a pointer pointing to the address of the start point in a partition with rank x
            - count - will be total number of columns * number of rows allocated for each partition
            - datatype - height is double so the expected data type should also be double,
            - destination - destination of every worker host should be the rank of master host (0),
            - tag - should be workers rank
        - While all the workers are sending the blocks of data, master should be aware from which rank it is receiving the requests. To keep track of the requests I have calculated values of start pointer and buffer size for each host based on the rank of the host
            - *"MPI_Recv(&(z[t%3][s][0]), size * rAllocated, MPI_DOUBLE, r, r, MPI_COMM_WORLD, &status);"*
        - Total blocks send/receive does not need to happen for time intervals where master node doesn't need to print wave heights. I optimized code to send/receive only when there is a need to output wave heights.

    - For exchange of edge rows between workers, algorithm I have written goes to each partition and worker with lower rank will first send (will be received by worker with higher rank) and vice versa happens.
    - This is done so that we avoid deadlocks, for example if two ranks want to send data to each other at the same time, then none of them would be ready to receive the data. Since MPI_Send and MPI_Recv are blocking calls, both the hosts will not be able to proceed further. This puts them in a deadlock.
    - Since each partition's IStart and IEnd are available, it is easier to construct MPI_Send and MPI_Receive calls.
- This whole process is run for t = 2 to (max_time) number of times.
- Output will be printed only when the rank is master and when the modulo of time t and interval equal zero.
- How did I parallelize this program using openMP?
    - For each time t, the calculation of one row is dependent on calculation of other row, which is a prime representation of static load balancing which I have implemented using *"#pragma omp for schedule(static)"*. I have not implemented parallelization on layers 0 and 1, as it would not decrease the output execution time significantly.

<u>Source code:</u> (*wave2d_mpi.cpp*)

```
#include "mpi.h"
#include <math.h>
#include <iostream>
#include "Timer.h"
#include <stdlib.h>   // atoi
#include "omp.h"

int constant = 10;
int default_size = 10*constant;  // the default system size
int defaultCellWidth = 8;
double c = 1.0;      // wave speed
double dt = 0.1;     // time quantum
double dd = 2.0;     // change in system
bool alwaysBlockSend = false;

using namespace std;

int main( int argc, char *argv[] ) {
    // verify arguments
    if ( argc != 5 ) {
        cerr << "usage: Wave2D size max_time interval" << endl;
        return -1;
    }
    int my_rank;                          // rank of process
    int numOfHosts;                       // total # of hosts
    int size = atoi( argv[1] );           //# of processes
    int max_time = atoi( argv[2] );       //# of times to calculate heights
    int interval  = atoi( argv[3] );      // when prints should happen
    int numOfThreads = atoi ( argv[4]); // total # of input threads
    omp_set_num_threads(numOfThreads);
    MPI_Status status;                    //receive

    if ( size < 100 || max_time < 3 || interval < 0 ) {
        cerr << "usage: Wave2D size max_time interval" << endl;
        cerr << "       where size >= 100 && time >= 3 && interval >= 0" << endl;
        return -1;
    }

    MPI_Init( &argc, &argv ); // start MPI
    MPI_Comm_rank(MPI_COMM_WORLD , &my_rank);      // find out process rank
    MPI_Comm_size(MPI_COMM_WORLD , &numOfHosts);   // find out # of processes

    // create a simulation space
    double z[3][size][size];
    for ( int p = 0; p < 3; p++ )
        for ( int i = 0; i < size; i++ )
            for ( int j = 0; j < size; j++ )
                z[p][i][j] = 0.0; // no wave

    // start a timer
    Timer time;
    time.start( );


    // time = 0;
    // initialize the simulation space: calculate z[0][i][j]
    int weight = size / default_size;
    for( int i = 0; i < size; i++ ) {
        for( int j = 0; j < size; j++ ) {
            if( i > 4 * weight * constant && i < 6 * weight * constant &&
                j > 4 * weight * constant && j < 6 * weight * constant)
            {
                z[0][i][j] = 20.0;
            }
            else
            {
                z[0][i][j] = 0.0;
            }
        }
    }

    //time = 1;
    // calculate the heights of cells at level 1 for t=1
    for (int i = 1 ; i < size-1 ; i++) {
        for (int j = 1 ; j < size-1 ; j++) {
            z[1][i][j] = z[0][i][j] + pow(c,2)/2 * (pow((dt/dd),2)) * (z[0][i+1][j]
+ z[0][i-1][j] + z[0][i][j+1] + z[0][i][j-1] - 4.0 * z[0][i][j]);
        }
    }

    // find the values of quotients and remainder
    int remainder = size % numOfHosts;
    int quotient =  size / numOfHosts;

    // initialize the values of iStart(start point of i), iEnd(end point of i) and
rowsAllocated
    int iStart, iEnd, rowsAllocated;

    iStart = my_rank * quotient;
    iStart += my_rank < remainder ? my_rank : remainder;

    iEnd = (my_rank + 1) * quotient;
    iEnd += my_rank < remainder ? my_rank + 1 : remainder;

    rowsAllocated = quotient;
    rowsAllocated += my_rank < remainder ? 1 : 0;

    //used to print the ranges of all the ranks
    cerr << "rank[" << my_rank << "]'s range = " << iStart << " ~ " << iEnd - 1 <<
endl;

    iStart += my_rank == 0 ? 1 : 0;
    iEnd -= my_rank == (numOfHosts - 1) ? 1 : 0;

    // we will run schrodingers wave equation parallelly on multiple hosts
    for (int t = 2; t < max_time; ++t ) {
        #pragma omp for schedule(static)
        for (int i = iStart ; i < iEnd ; i++ ) {
            for (int j = 1 ; j < size - 1 ; j++) {
                z[t%3][i][j] = 2.0 * z[(t-1)%3][i][j]
                                - z[(t-2)%3][i][j]
                                + pow(c,2)*(pow((dt/dd),2))
                                * (
                                   z[(t-1)%3][i+1][j]
                                   + z[(t-1)%3][i-1][j]
                                   + z[(t-1)%3][i][j+1]
                                   + z[(t-1)%3][i][j-1]
                                   - 4.0 * z[(t-1)%3][i][j]
                                  );
            }
        }

        /*
         * if current host is master then master will calculate
         * the start point and ranks of all the partitions and use the
         * calculated values to findout the height at specific point
         */

        if (my_rank == 0)
        {
            if (alwaysBlockSend || (interval != 0 && t % interval == 0))
            {
                for (int r = 1; r < numOfHosts; r++)
                {
                    int rAllocated = quotient;
                    if (r < remainder) {
                        rAllocated += 1;
                    }
                    int s;
                    if (r < remainder)
                    {
                        s = (r * quotient) + r;
                    }
                    else {
                        s = r * quotient + remainder;
                    }
                    MPI_Recv(&(z[t%3][s][0]), size * rAllocated, MPI_DOUBLE, r, r,
MPI_COMM_WORLD, &status);
                }
            }

        }
        /* if current host is a worker host other than master then,
         * the host will send the calculated block of data back to master
         */
        else
        {
            if (alwaysBlockSend || (interval != 0 && t % interval == 0))
            {
                MPI_Send(&(z[t%3][iStart][0]), size * rowsAllocated, MPI_DOUBLE, 0,
my_rank, MPI_COMM_WORLD );
            }
        }

        for (int partition = 0; partition < numOfHosts - 1; partition++)
        {
            /* when rank equals partition, then rank will send
             * last row in its partition to rank+1, and will receive
             * first row from rank+1
             */
            if (partition == my_rank)
            {
                MPI_Send(&(z[t%3][iEnd-1][0]), size, MPI_DOUBLE, my_rank + 1,
my_rank, MPI_COMM_WORLD );

                MPI_Recv(&(z[t%3][iEnd][0]), size, MPI_DOUBLE, my_rank + 1, my_rank
+ 1,  MPI_COMM_WORLD, &status);
            }

            /* when rank is one less than partition number, then
             * first we will receive last row from rank - 1 and
             * later we will send first row to rank - 1
             */
            if (partition == (my_rank - 1))
            {
                MPI_Recv(&(z[t%3][iStart - 1][0]), size, MPI_DOUBLE, my_rank - 1,
my_rank - 1,  MPI_COMM_WORLD, &status);

                MPI_Send(&(z[t%3][iStart][0]), size, MPI_DOUBLE, my_rank - 1,
my_rank, MPI_COMM_WORLD );

            }
        }

        /* master host will print output only when interval is not 0
         * and when mod of time and interval is 0
         */
        if (my_rank == 0) {
                if (interval != 0 && t % interval == 0)
                {
                        cout << t << endl;
                        for (int i = 0 ; i < size ; i++) {
                                for(int j = 0; j < size ; j++) {
                                        cout << z[t%3][j][i] << " ";
                                }
                        cout << endl;
                        }
                }
        }
    }
  if(my_rank == 0)
{
    cerr << "Elapsed time = " << time.lap( ) << endl;
}

  MPI_Finalize( ); // Shut down MPI
  return 0;
 } // end of simulation
```

Execution output :



```
[svrb@cssmpi2h prog2]$ vi wave2d_mpi.cpp
[svrb@cssmpi2h prog2]$ mpic++ wave2d_mpi.cpp Timer.cpp -fopenmp -o wave2d_mpi; mpiexec -n 6 ./wave2d_mpi 100 500 10 4 > wave2d_mpi
_output.txt; diff out_sequential.txt wave2d_mpi_output.txt
rank[0]'s range = 0 ~ 16
rank[1]'s range = 17 ~ 33
rank[2]'s range = 34 ~ 50
rank[3]'s range = 51 ~ 67
rank[4]'s range = 68 ~ 83
rank[5]'s range = 84 ~ 99
Elapsed time = 2179895
[svrb@cssmpi2h prog2]$ mpirun -np 1 ./wave2d_mpi 588 500 0 1
rank[0]'s range = 0 ~ 587
Elapsed time = 5894149
[svrb@cssmpi2h prog2]$ mpirun -np 6 ./wave2d_mpi 588 500 0 1
rank[0]'s range = 0 ~ 97
rank[5]'s range = 490 ~ 587
rank[3]'s range = 294 ~ 391
rank[1]'s range = 98 ~ 195
rank[4]'s range = 392 ~ 489
rank[2]'s range = 196 ~ 293
Elapsed time = 1918642
[svrb@cssmpi2h prog2]$ mpirun -np 6 ./wave2d_mpi 588 500 0 4
rank[0]'s range = 0 ~ 97
rank[1]'s range = 98 ~ 195
rank[5]'s range = 490 ~ 587
rank[3]'s range = 294 ~ 391
rank[4]'s range = 392 ~ 489
rank[2]'s range = 196 ~ 293
Elapsed time = 1919550
[svrb@cssmpi2h prog2]$ 
```

From the above output we can say that wave2d_mpi has been compiled successfully and there is no difference in output between sequential execution and parallel execution with Mpi and OpenMp.

- Runtime for one host with single thread took around 2179895 time
- Performance improvement ratio of one machine to 6 machines with single thread is (5894149 / 1918642) = 3.072 times faster.
- Performance improvement ratio of one machine to 6 machines with 4 threads is (5894149 / 1919550) = 3.070 times faster.

Discussion on assignment 2

To calculate the Schrodinger's wave equation we need time t, i position and j position. Using this equation, we have calculated the height of each block in an (n x n) matrix. In which way did i calculate? we must do a lot of computation to calculate the height at each block every single time, to implement this on single host is time killing. So, I have made use of the unused remote hosts to calculate these values. How did I share the load? I have shared the load equally amongst the available hosts by dividing the size of matrix by total number of hosts, if there remains any remainder number of rows then those remainder rows will be added to hosts which have rank lesser than the remainder.
I implemented MPI parallelization among all the partitions and parallelized it using static load balancing, which has reduced the running elapsed time by a significant amount.

<u>Areas to be Improved:</u>

- So, I implemented MPI_Send and MPI_Recv where I realized that until the send req from rank x is received by rank y, rank x stays in blocked state which causes a lot of delay so to avoid this situation I tried implementing MPI_iSend and MPI_iRecv which don't wait for any acknowledgement and continue with their process. For iSend and iRecv i had to explicitly implement MPI_wait() function to check if the communication tokk place properly or not which I tried using in my code, but that resulted in multiple complications and I encountered one by one error. So, I reverted my changes and completed sending and receiving using MPI_Send and MPI_Recv.
- I could have implemented parallelization on level 1 i.e.. where t = 1, as it is only t number of runs, I assumed that the execution time would not drastically change/ fall down.

<u>Limitations</u>:

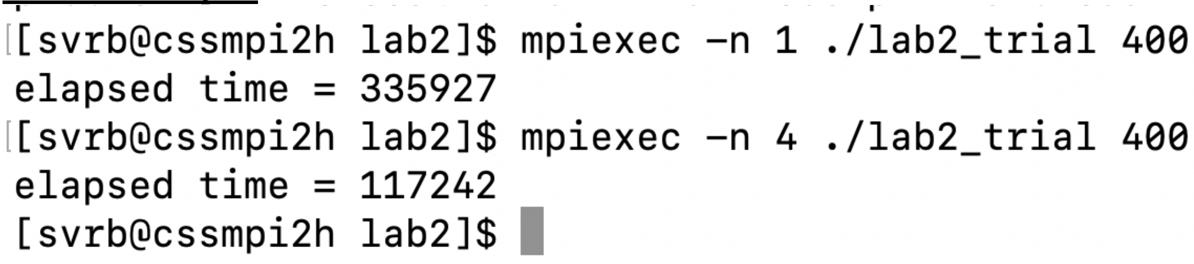
- One of the positive limitations in my algorithm is that, instead of sending blocks of calculated data after every single execution (which would cause unnecessary delay in time) I have avoided that by adding a check where, if mod of time and interval equals zero then only workers will send the data and master will receive it.
- I remember in previous assignment it was mentioned that the total number of cores on MPI machines are three and only 3 threads would yield better output. For *"mpirun -np x ./wave2d_mpi 588 500 0 3*" I am able to produce better output than for 4 threads. So, I assume that as the number of threads go beyond total number of cores then the outputs will be degrading.

Parallel Programming Lab2 Report

<u>Matrix_mpi source code</u> (lab2_trial.cpp) :

```
#include "mpi.h"
#include <stdlib.h> // atoi
#include <iostream> // cerr
#include "Timer.h"

using namespace std;

void init( double *matrix, int size, char op ) {
  for ( int i = 0; i < size; i++ )
    for ( int j = 0; j < size; j++ )
      matrix[i * size + j]
        = ( op == '+' ) ? i + j :
        ( ( op == '-' ) ? i - j : 0 );
}

void print( double *matrix, int size, char id ) {
  for ( int i = 0; i < size; i++ )
    for ( int j = 0; j < size; j++ )
      cout << id << "[" << i << "][" << j << "] = " << matrix[i * size + j] << endl;
}

void multiplication( double *a, double *b, double *c, int stripe, int size ) {
  for ( int k = 0; k < size; k++ )
    for ( int i = 0; i < stripe; i++ )
      for ( int j = 0; j < size; j++ )
        // c[i][k] += a[i][j] * b[j][k];
        c[i * size + k] += a[i * size + j] * b[j * size + k];
}

int main( int argc, char* argv[] ) {
  int my_rank = 0;            // used by MPI
  int mpi_size = 1;           // used by MPI
  int size = 400;             // array size
  int tag;
  bool print_option = false;  // print out c[] if it is true
  Timer timer;
  MPI_Status status;

  // variables verification
  if ( argc == 3 ) {
    if ( argv[2][0] == 'y' )
      print_option = true;
  }

  if ( argc == 2 || argc == 3 ) {
    size = atoi( argv[1] );
  }
  else {
    cerr << "usage:   matrix size [y|n]" << endl;
    cerr << "example: matrix 400   y" << endl;
    return -1;
  }

  MPI_Init( &argc, &argv ); // start MPI
  MPI_Comm_rank( MPI_COMM_WORLD, &my_rank );
  MPI_Comm_size( MPI_COMM_WORLD, &mpi_size );

  // matrix initialization
  double *a = new double[size * size];
  double *b = new double[size * size];
  double *c = new double[size * size];

  if ( my_rank == 0 ) { // master initializes all matrices
    init( a, size, '+' );
    init( b, size, '-' );
    init( c, size, '0' );

    // print initial values
    if ( false ) {
      print( a, size, 'a' );
      print( b, size, 'b' );
    }

    // start a timer
    timer.start( );
  }
  else {                // slavs zero-initializes all matrices
    init( a, size, '0' );
    init( b, size, '0' );
    init( c, size, '0' );
  }

  // broadcast the matrix size to all.
  MPI_Bcast( &size, 1, MPI_INT, 0, MPI_COMM_WORLD );

  int stripe = size / mpi_size;     // partitioned stripe

  if(my_rank == 0) {
  for (int worker = 1 ; worker < mpi_size ; worker++) {

        // master sends each partition of a[] to a different slave
        MPI_Send(a+stripe*size*worker, stripe*size , MPI_DOUBLE, worker, 0 ,MPI_COMM_WORLD);

        // master also sends b[] to all slaves
        MPI_Send(b, size*size , MPI_DOUBLE, worker, 0 ,MPI_COMM_WORLD);
   }
  }
  else { // for workers
        MPI_Recv(a, stripe*size , MPI_DOUBLE, 0 , 0 , MPI_COMM_WORLD , &status);
        MPI_Recv(b, size*size , MPI_DOUBLE , 0 , 0 ,MPI_COMM_WORLD, &status);
  }

  multiplication( a, b, c, stripe, size ); // all ranks should compute multiplication

  // master receives each partition of c[] from a different slave
        if(my_rank == 0) {
        for (int worker = 1 ; worker < mpi_size ; worker++) {
          MPI_Recv(c+stripe*size*worker , stripe*size , MPI_DOUBLE, worker, 0 ,MPI_COMM_WORLD
, &status);
                }
        }
        else {
                MPI_Send( c, stripe*size , MPI_DOUBLE, 0 , 0 , MPI_COMM_WORLD);
        }

  if ( my_rank == 0 )
    // stop the timer
    cout << "elapsed time = " << timer.lap( ) << endl;

  // results
  if ( print_option && my_rank == 0 )
    print( c, size, 'c' );

   MPI_Finalize( ); // shut down MPI
}
```

Execution Output:



```
[svrb@cssmpi2h lab2]$ mpiexec -n 1 ./lab2_trial 400
elapsed time = 335927
[svrb@cssmpi2h lab2]$ mpiexec -n 4 ./lab2_trial 400
elapsed time = 117242
[svrb@cssmpi2h lab2]$
```

Observations:

My observation during execution of this matrix multiplication program was that parallelizing the hosts will yield better results than serialization. How? Well, assume that we are working on a quadcore computer, if we write program in a serialized manner, then the whole computation load will fall only on one core and the execution takes more time and other cores/hosts will be idle (we can check the elapsed time for 1 thread). So, instead of wasting more time on execution, we can improve the running time by making use of multiple cores where we can split the same job between different hosts. MPI will be used to split the jobs between different hosts, one host will act like a master and split the tasks between other cores/workers using MPI_Send(). Workers will have the same code that master has and will receive information from master by using MPI_Recv(), workers will implement the code according to their rank numbers. In this way most of the cores can be utilized and the same job can be completed in lesser time. This was my observation on matrix multiplication using mpi.

- ***Srilekha Venkata Ramani Bandaru***