Programming 2 Assignment Report

Student Name: Dazhi Li

Student NETID: dazhili

Student ID#: 2400330

# 1. Documentation

This part briefly describes how I implement my Wave2D algorithms, MPI communication and OMP strategies.

## Wave2D Diffusion Algorithms

The wave2D diffusion algorithms is exactly the same as Schroedinger's wave dissemination formula.

In t=0, setup an initial wave at the center of the graph, where x axis and y axis are between 0.4 size to 0.6 size.

In t=1 and t =2, calculating the whole graph by Schroedinger's wave dissemination function. Only notice that the edge of the graph is not included for the computation for leaving them being 0 all the time. Make a for loop just starts from index=1 and ends at index = size -2.

For the rotation part, I utilized the remainder of each iteration:

Now calculating = z[t%3], Previously = z[(t-1)%3], Previously previous= z[(t-2)%3]

## MPI Communication

I divided my wave matrix into strips by number of working nodes. If the size could be exactly divided by the number of working nodes, the workload is equally distributed. However, if the size cannot be divided by the number of working nodes and leaves a remainder, those nodes with smaller ranks will consume the remainder like adding one row into each node’s workload.

102 size matrix could be distributed to 4 nodes like:

[rank0] = 25 + 1(from remainder) = 26

[rank1] = 25 + 1(from remainder) = 26

[rank3] = 25

[rank4] = 25

After the algorithm of job assigning, I would like to discuss about how the graph should be allocated. In my opinion, the graph should be allocated horizontally to each nodes, not vertically allocated. The reason is quite simple. When we are using MPI\_Send(), we want to send a message which is **continuously allocated** **in the RAM**. The vertical allocating methods sends data that are distributed in the RAM. The difference are shown in Fig1.

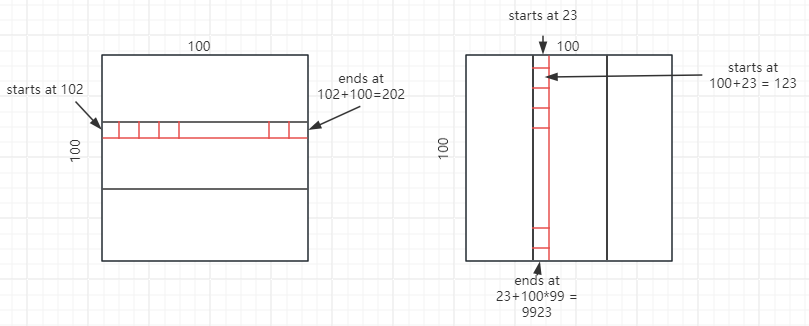
I am not saying that the vertical allocation is wrong. However, horizontal allocation suits this problem better and shows a shorter Loc.

Fig1. Horizontal allocation vs Vertical allocation

Finally, I would like to discuss about the communication ways in my program. Notice that when t=0, t>=2 is different.

When t=0, master node will initialize the wave in the center of the graph and send data to the related nodes, as shown in fig2.

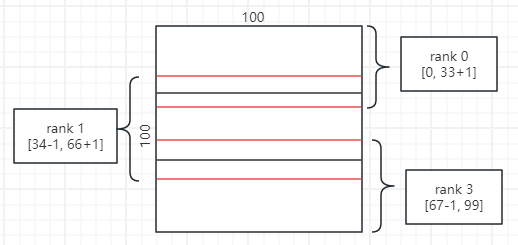


Fig2. Allocation when t=0

When the first-time communication finished, each node could compute the graph at t=1.

When it comes to t=2 and later iteration, each node communicates first. Then they do their computation. This time the communication is different.

Starting from the first rank, each node sends their message to the next node except the last node, because the last node has no next node after it.

Then, each node begins to receive message from their previous node except the master node for there is no previous node for it.

After that, these nodes begin to send information to their previous node.

Finally, all the nodes begin to receive message from their next node except the last node.

The whole process is shown in fig3.

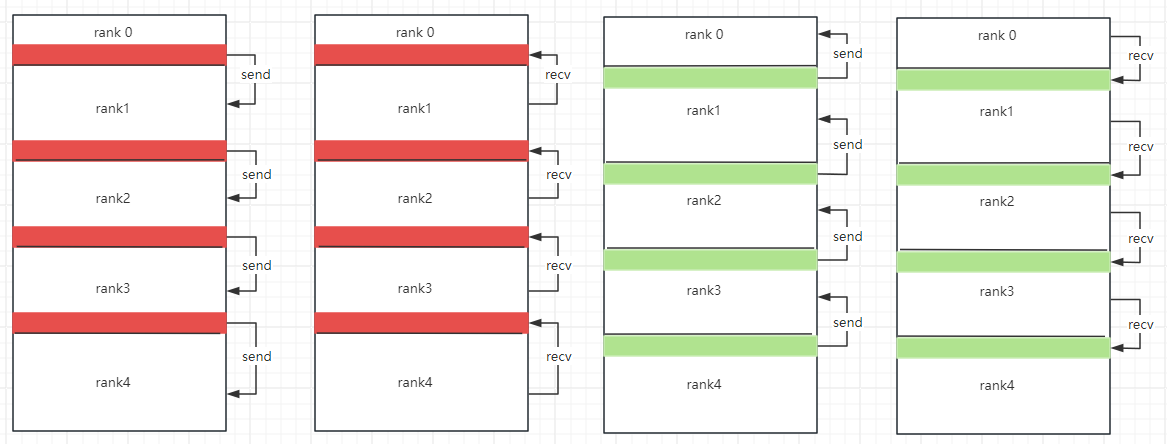


Fig3. Communication on neighboring rows when t>=2

As you can see, each neighboring row between two nodes are sent and received once.

As for the printing strategy, there is no need to talk about the details as it is extremely simple. When interval is not zero and it is just the time to print, all salve nodes send their massage, which contains only the working field of each node, to the master node. After receiving all the messages from the slave nodes, master node prints out the graph by going through it.

## OpenMP strategies

My open MP strategy just focuses on the Schroedinger's wave diffusion calculation part. When t=1, each node will begin their computation of z[1][start\_ixd] to z[1][end\_idx] with openMP. The parallel is deployed right in front of this for loop.

When t>=2, the OMP parallel strategy is quite the same as t=1. I utilize OMP to calculating the z[t%3], which is the main time cost in my program.

# 2.Source Code

I will show the Wave2D.cpp first, which is the non-paralleled version of computing wave diffusion by Schroedinger's formula.

Then come up with the Wave2D\_mpi.cpp, which implements MPI and OMP paralleling strategies.

Please note: all the codes are accessible through the attached zip file.

**Wave 2D.cpp**

#include <iostream>

#include "Timer.h"

#include <stdlib.h>   // atoi

int default\_size = 100;  // 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

using namespace std;

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

  // verify arguments

  if ( argc != 4 ) {

    cerr << "usage: Wave2D size max\_time interval" << endl;

    return -1;

  }

  int size = atoi( argv[1] );

  int max\_time = atoi( argv[2] );

  int interval  = atoi( argv[3] );

  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;

  }

  // 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][][]

  int weight = size / default\_size;

  for( int i = 0; i < size; i++ ) {

    for( int j = 0; j < size; j++ ) {

      if( i > 40 \* weight && i < 60 \* weight  &&

    j > 40 \* weight && j < 60 \* weight ) {

        z[0][i][j] = 20.0;

      } else {

        z[0][i][j] = 0.0;

      }

    }

  }

  // time = 1

  // calculate z[1][][]

  // cells not on edge

  // IMPLEMENT BY YOURSELF !!!

  for (int i = 1; i < size-1; i++) {

    for (int j = 1; j < size-1; j++) {

      z[1][i][j] = z[0][i][j] + c\*c/2\*(dt/dd)\*(dt/dd)\*(z[0][i+1][j]+z[0][i-1][j]+z[0][i][j+1]+z[0][i][j-1]-4\*z[0][i][j]);

    }

  }

  // simulate wave diffusion from time = 2

  for ( int t = 2; t < max\_time; t++ ) {

    for (int i = 1; i < size - 1; i++) {

      for (int j = 1; j < size - 1; j++) {

        z[t%3][i][j] = 2\*z[(t-1)%3][i][j] - z[(t-2)%3][i][j] + c\*c\*(dt/dd)\*(dt/dd)\*

        (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\*z[(t-1)%3][i][j]);

      }

    }

    // printing out by interval

    if (interval !=0 && (t % interval == 0 || t == max\_time - 1)) {

      cout << t << endl;

      for (int i = 0; i < size; i++) {

        for (int j = 0; j < size; j++) {

          cout << z[t%3][i][j] << " ";

        }

        cout << endl;

      }

    }

  } // end of simulation

  // finish the timer

  cerr << "Elapsed time = " << time.lap( ) << endl;

  return 0;

}

**Wave2D\_mpi.cpp**

#include <iostream>

#include "Timer.h"

#include <stdlib.h>   // atoi

#include "mpi.h"

#include "omp.h"

int default\_size = 100;  // 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

int my\_rank = 0;            // used by MPI

int mpi\_size = 1;           // used by MPI

MPI\_Status status;

using namespace std;

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

  // verify arguments

  if ( argc != 4 && argc != 5 ) {

    cerr << "usage: Wave2D size max\_time interval num\_thread(optional)" << endl;

    return -1;

  }

  int size = atoi( argv[1] );

  int max\_time = atoi( argv[2] );

  int interval  = atoi( argv[3] );

  int num\_thread = 1;

  if (argc == 5) {

    num\_thread = atoi( argv[4] );

  }

  if ( size < 100 || max\_time < 3 || interval < 0 || num\_thread < 1) {

    cerr << "usage: Wave2D size max\_time interval" << endl;

    cerr << "       where size >= 100 && time >= 3 && interval >= 0 && #thread >=1" << endl;

    return -1;

  }

  // 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

  MPI\_Init( &argc, &argv ); // start MPI

  MPI\_Comm\_rank( MPI\_COMM\_WORLD, &my\_rank );

  MPI\_Comm\_size( MPI\_COMM\_WORLD, &mpi\_size );

  omp\_set\_num\_threads( num\_thread );

  // Allocating strips for each node

  int strips[mpi\_size], start\_idx[mpi\_size], end\_idx[mpi\_size];

  start\_idx[0] = 0;

  int remainder = size % mpi\_size, standard\_strip = size / mpi\_size;

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

    strips[i] = standard\_strip;

  }

  if (remainder != 0) {

    for (int i = 0; i < remainder; i++) {

      strips[i] += 1;

    }

  }

  for (int i = 1; i < mpi\_size; i++) {

    start\_idx[i] = start\_idx[i-1] + strips[i-1];

    end\_idx[i-1] = start\_idx[i] - 1;

  }

  //specialize first and last node

  start\_idx[0] = 1;

  end\_idx[mpi\_size - 1] = size - 2;

  Timer time;

  time.start( );

  // master initializing t=0

  if (my\_rank == 0) {

    // start a timer

    // time = 0;

    // initialize the simulation space: calculate z[0][][]

    int weight = size / default\_size;

    for( int i = 0; i < size; i++ ) {

      for( int j = 0; j < size; j++ ) {

        if( i > 40 \* weight && i < 60 \* weight  &&

        j > 40 \* weight && j < 60 \* weight ) {

        z[0][i][j] = 20.0;

      } else {

        z[0][i][j] = 0.0;

      }

      }

    }

    // send message to nodes

    if (mpi\_size > 1) {

      for (int i = 1; i < mpi\_size - 1; i++) {

        MPI\_Send(z[0][end\_idx[i-1]], size\*(strips[i] + 2), MPI\_DOUBLE, i, 0, MPI\_COMM\_WORLD);

      }

        MPI\_Send(z[0][end\_idx[mpi\_size-2]], size\*(strips[mpi\_size-1] + 1),

         MPI\_DOUBLE, mpi\_size - 1, 0, MPI\_COMM\_WORLD);

    }

  } else {

    if (my\_rank != mpi\_size -1) {

      MPI\_Recv(z[0][end\_idx[my\_rank-1]], size\*(strips[my\_rank] + 2),

       MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD, &status);

    } else {

      MPI\_Recv(z[0][end\_idx[my\_rank-1]], size\*(strips[my\_rank] + 1),

       MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD, &status);

    }

  }

  // time = 1

  // calculate z[1][][]

  // cells not on edge

#pragma omp parallel for

  for (int i = start\_idx[my\_rank]; i <= end\_idx[my\_rank]; i++) {

    for (int j = 1; j < size-1; j++) {

      z[1][i][j] = z[0][i][j] + c\*c/2\*(dt/dd)\*(dt/dd)\*(z[0][i+1][j]+z[0][i-1][j]+z[0][i][j+1]+z[0][i][j-1]-4\*z[0][i][j]);

    }

  }

  // simulate wave diffusion from time = 2

  for ( int t = 2; t < max\_time; t++ ) {

    // message exchanging

    if (mpi\_size > 1) {

      if (my\_rank != mpi\_size - 1) {

        MPI\_Send(z[(t-1)%3][end\_idx[my\_rank]], size, MPI\_DOUBLE, my\_rank + 1, 0, MPI\_COMM\_WORLD);

      }

      if (my\_rank != 0) {

        MPI\_Recv(z[(t-1)%3][end\_idx[my\_rank - 1]], size, MPI\_DOUBLE, my\_rank - 1, 0, MPI\_COMM\_WORLD, &status);

        MPI\_Send(z[(t-1)%3][start\_idx[my\_rank]], size, MPI\_DOUBLE, my\_rank - 1, 0, MPI\_COMM\_WORLD);

      }

      if (my\_rank != mpi\_size -1) {

        MPI\_Recv(z[(t-1)%3][start\_idx[my\_rank+1]], size, MPI\_DOUBLE, my\_rank + 1, 0, MPI\_COMM\_WORLD, &status);

      }

    }

    // calculating

  #pragma omp parallel for

    for (int i = start\_idx[my\_rank]; i <= end\_idx[my\_rank]; i++) {

      for (int j = 1; j < size - 1; j++) {

        z[t%3][i][j] = 2\*z[(t-1)%3][i][j] - z[(t-2)%3][i][j] + c\*c\*(dt/dd)\*(dt/dd)\*

        (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\*z[(t-1)%3][i][j]);

      }

    }

    // printing out by interval

    if (interval !=0 && (t % interval == 0 || t == max\_time - 1)) {

      // exchanging data first

      if (mpi\_size > 1) {

        if (my\_rank != 0) {

          MPI\_Send(z[t%3][start\_idx[my\_rank]], size\*strips[my\_rank], MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD);

        } else {

          for (int i = 1; i < mpi\_size; i++) {

            MPI\_Recv(z[t%3][start\_idx[i]], size\*strips[i], MPI\_DOUBLE, i, 0, MPI\_COMM\_WORLD, &status);

          }

        }

      }

      // printing

      if (my\_rank == 0) {

        cout << t << endl;

        for (int i = 0; i < size; i++) {

          for (int j = 0; j < size; j++) {

            cout << z[t%3][i][j] << " ";

          }

          cout << endl;

        }

      }

    }

  } // end of simulation

  // finish the timer

  if (my\_rank == 0) {

    cerr << "Elapsed time = " << time.lap( ) << endl;

  }

  MPI\_Finalize( ); // shut down MPI

  return 0;

}

# 3.Execution Result

First, I will visualize the output of the wave diffusion as shown in fig4.

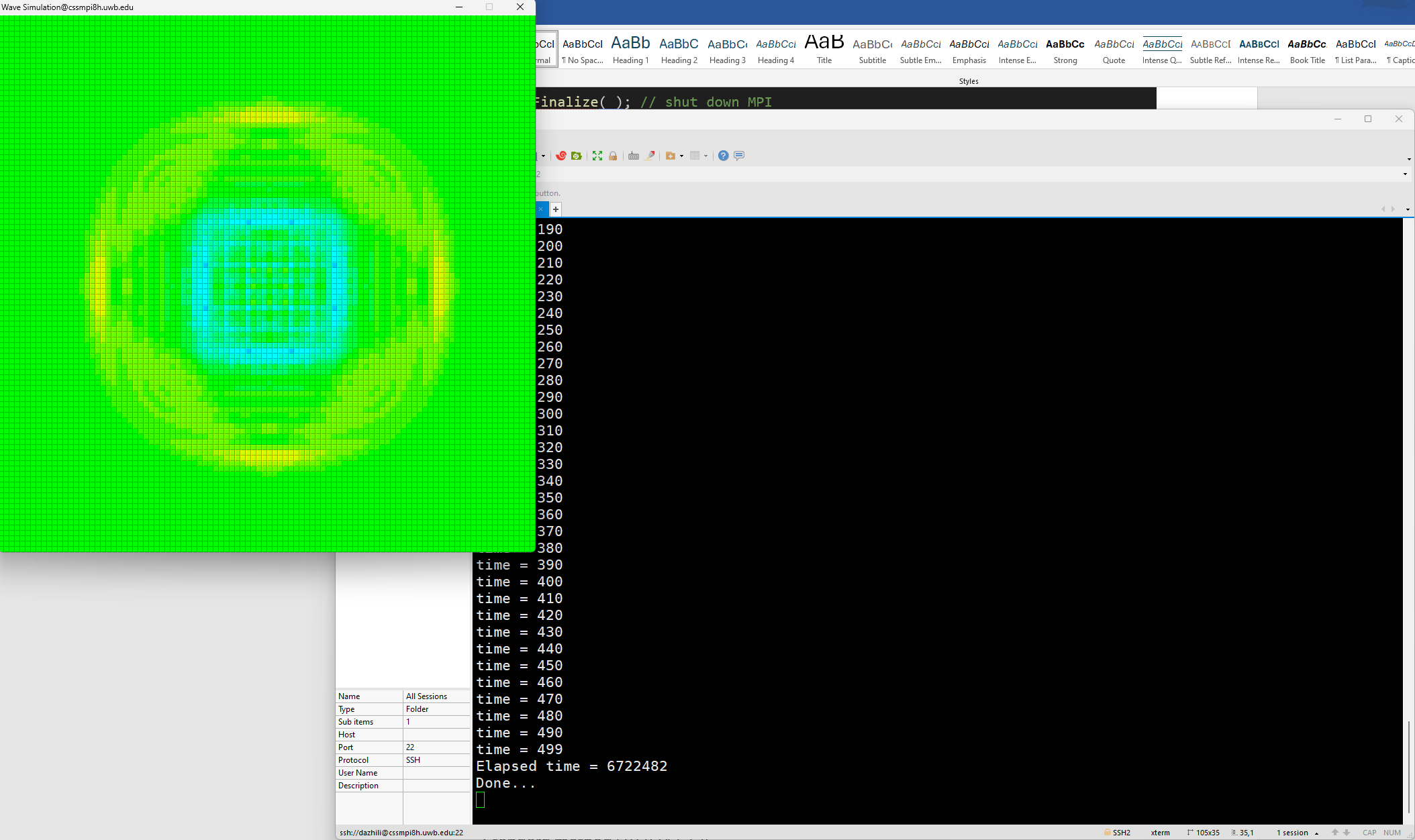


Fig4. Wave2D visualized result

I am not comparing my out1.txt with the professor’s out1.txt for one reason. I noticed that the professor’s output initializing the wave at t=3. However, we are required to accomplish this initialization at t=0. Professor’s output is full 0 when t=2 which means that the initialization process has not started yet. And by comparing my out1.txt with the professors’ when interval = 1, I found that there is 3 steps shift between ours as shown in fig5. But I am certain that this difference is not the core of this program. So, I just sticked on my own way, which generates different output to the professor’s.

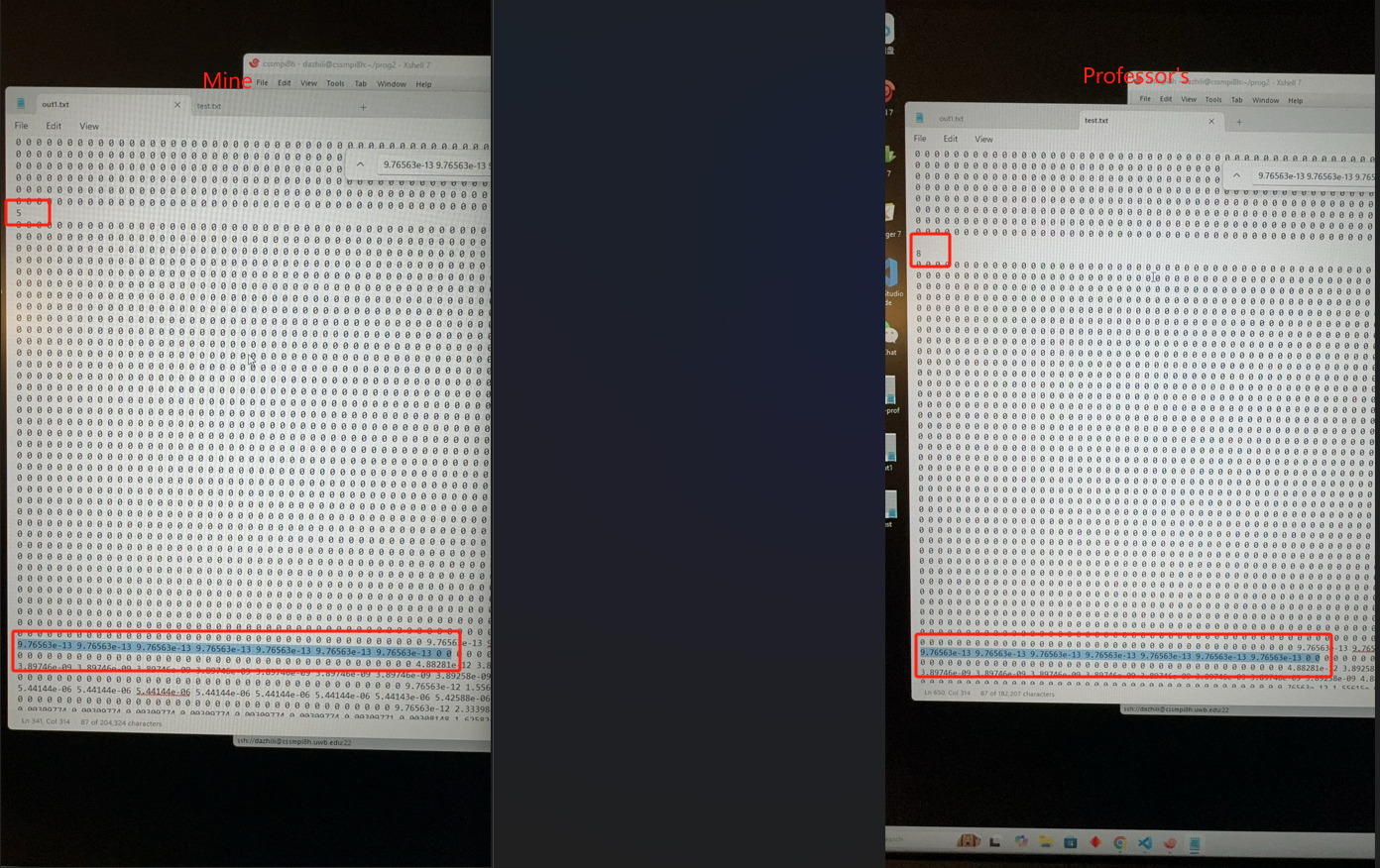


Fig5. 3 time shifts between my output and professor’s

Second, the difference between the output of my Wave2D.cpp and Wave2D\_mpi.cpp is shown below in fig6. Those two outputs are identical.

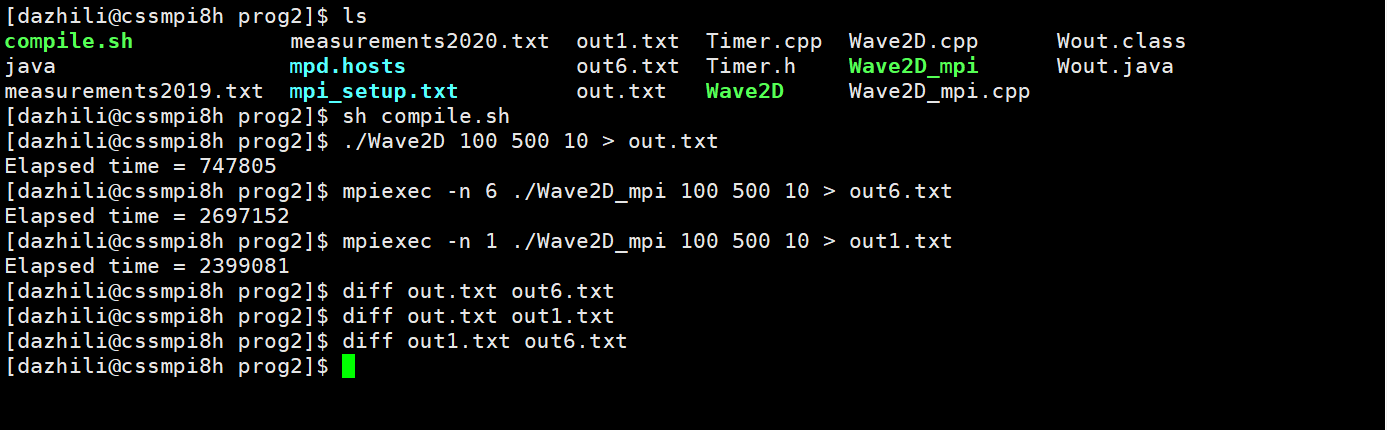


Fig6. Output difference between Wave2D and Wave2D\_mpi

Out.txt stands for the non-paralleled version output while out1.txt stands for mpi version of 1 machine and out6.txt stands for mpi version of 6 machines.

Third, the comparison between MPI=1, OMP=1 and MPI=6, OMP=1 is shown in fig7.

MPI=1, OMP=1: Elapsed time = 4173597

MPI=6, OMP=1: Elapsed time = 1551170

The improvement is 4173597/1551170= 2.69

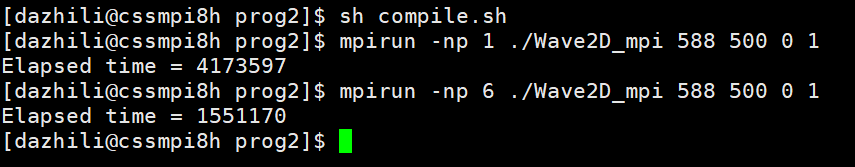


Fig7. Performance Comparison between MPI=1 and MPI=6 when OMP=1

Finally, the comparison between MPI=1, OMP=1 and MPI=6, OMP=4 is shown in fig8.

MPI=1, OMP=1: Elapsed time = 4189585

MPI=6, OMP=4: Elapsed time = 977946

The improvement is 9592897/3371693 = 4.28

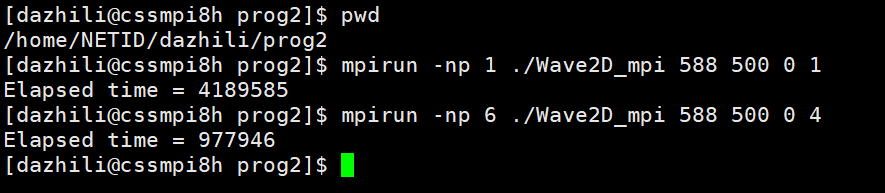


Fig8. Performance comparison between MPI=1, OMP=1 and MPI=6, OMP=4

# 4.Discussions

## 4.1 Hybrid MPI and OMP performance

This performance was measured at morning of 10/24/2024, when all the cssmpi machines were reset. The data includes number of machine (MPI) from 1 to 6 while number of threads (OMP) from 1 to 4. The results are shown in the chart below.

It is easy to conclude that with more machines and more threads running on each machine, the performance of the program is getting better and better. That satisfies what we have learnt in classes. However, there are two points that should be noticed. When number of threads running on per machine is 3, the performance is getting worse. That is due to the virtual cpu core communication layout as we have discussed in Program 1 and in class. The second point is when MPI = [3, 4], OMP=2 is even faster than OMP=4. This is possible due to some zombie process and the virtual cpu core layout. But most of the time, OMP=2’s performance is not better than OMP=4’s. So, I conclude it as environmental related bias in this experiment. After that, I tried running program under OMP=2 and OMP=4 when MPI = [3,4] a few times. Most of the time OMP=4 performed better.

## 4.2 How zombie process influences the performance

Before all the cssmpi machine were reset, I tried running the same program on cssmpi8h-cssmpi13h. The result is very weird as shown in fig 9.

This is easy to find that under the same number of machines = 6, OMP=2 truly improved the program performance while OMP=4 gets even slower than OMP=1. At this moment, I realized that my OMP strategy did work. However, after discussing with the professor after class, I conclude two reasons for that abnormal result. The first and most important reason is zombie process, which will directly influence the performance of OMP. The second reason is the virtual cpu core layout as I discussed above.

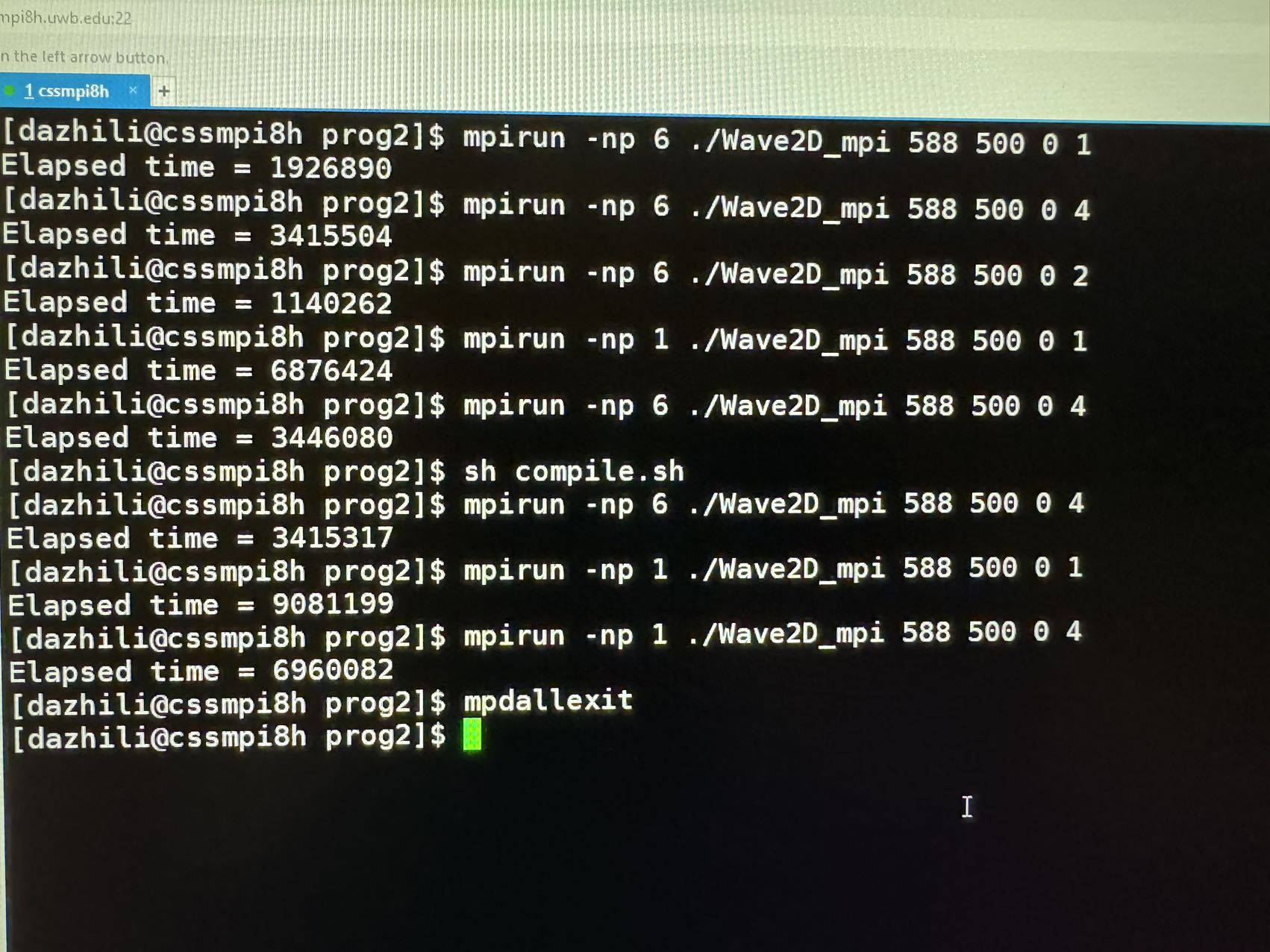


Fig9. Performance under influence of zombie process

# 5.LAB2 Details

## 5.1 Implementation

I implemented mpi on the matrix.cpp to get my own program matrix\_mpi.cpp

I utilized 8 machines to do this lab, including cssmpi8h~cssmpi15h.

The size of the matrix should be the dividable by 8. The whole process will running like what is shown in fig10.

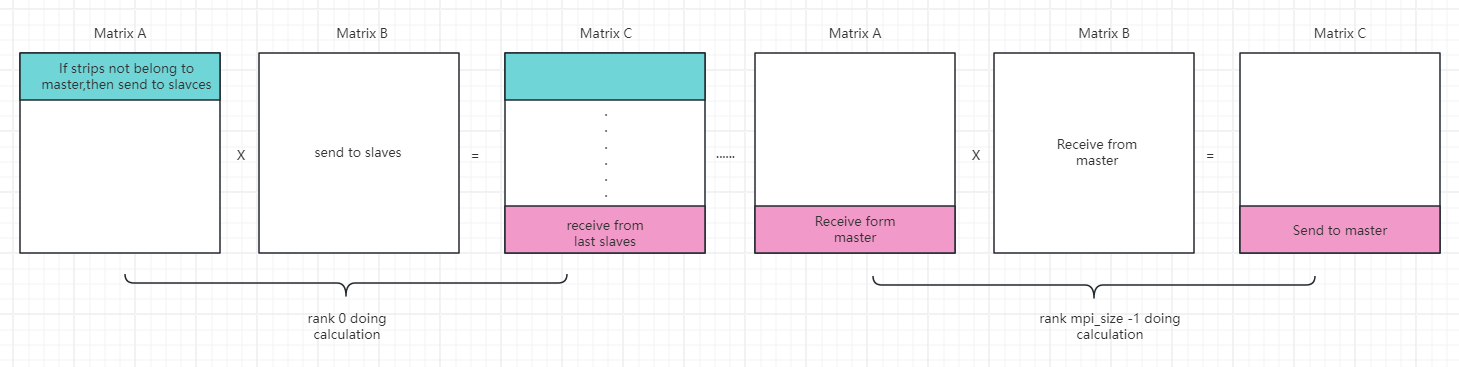


Fig10. Lab2 Programming MPI strategy

Source code:

#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

  bool print\_option = false;  // print out c[] if it is true

  MPI\_Status status;

  Timer timer;

  // 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 );

  // broadcast the matrix size to all.

  MPI\_Bcast( &size, 1, MPI\_INT, 0, MPI\_COMM\_WORLD );

  // 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' );

  }

  int stripe = size / mpi\_size;     // partitioned stripe

  // master sends each partition of a[] to a different slave

  if (my\_rank == 0) {

    for (int i=1; i<mpi\_size;i++) {

      double \*partition\_a = a + stripe\*size\*i;

      MPI\_Send(partition\_a, stripe\*size, MPI\_DOUBLE, i, 0, MPI\_COMM\_WORLD);

    }

  } else {

    MPI\_Recv(a, stripe\*size, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD, &status);

  }

  // master also sends b[] to all slaves

  if (my\_rank == 0) {

    for (int i=1; i<mpi\_size;i++) {

      MPI\_Send(b, size\*size, MPI\_DOUBLE, i, 0, MPI\_COMM\_WORLD);

    }

  } else {

    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) {

    MPI\_Send(c, stripe\*size, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD);

  } else {

    for (int i=1; i<mpi\_size; i++) {

      MPI\_Recv(c+stripe\*size\*i, stripe\*size, MPI\_DOUBLE, i, 0, MPI\_COMM\_WORLD, &status);

    }

  }

  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

}

## 5.2 Execution Result

As shown in Fig11.

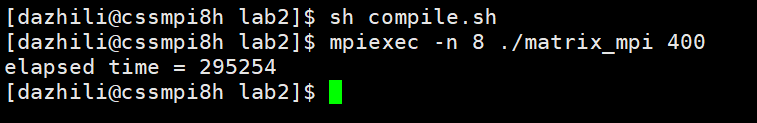


Fig11. Lab2 running result

If I choose to print out the matrix, the last row is like shown in Fig12.

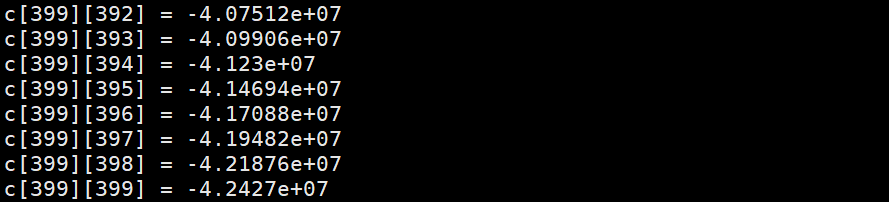


Fig12. Printed matrix of Lab2

# 6.Summary

In this report I briefly documented the Wave2D algorithms, OMP implementation and MPI parallelism strategies in my program 2. Also, MPI implementation in Lab2 is also included. Moreover, the performance of hybrid MPI and OMP are discussed above and related elements that could influence that performance.