CSS 534
Program 2: Parallelizing Wave Diffusion with MPI and OpenMP

Professor: Munehiro Fukuda

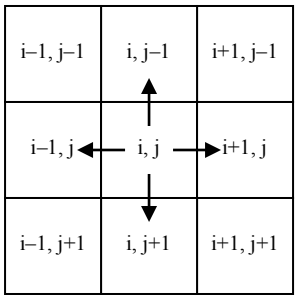
Due date: see the syllabus

1. Purpose

In this programming assignment, we will parallelize a sequential version of a two-dimensional wave diffusion program, using a hybrid form of MPI and OpenMPI.

2. Schroedinger's Wave Dissemination

Assume the water surface in a two-dimensional square bucket. To simulate wave dissemination over this water surface, let's partition this square in mesh and thus into N-by-N cells. Each cell(i, j) where 0 < i, j < N-1 maintains the height of its water surface. A wave is disseminated north, east, south, and west of each cell, and therefore cell(i, j) computes its new surface height from the previous height of itself and its four neighboring cells: cell(i+1, j), cell(i-1, j), cell(i, j+1) and cell(i, j-1). Let ***Zt_i,j***, ***Zt-1_i,j***, and ***Zt-2_i,j*** be the surface height of cell(i, j) at time t, time t-1, and time t-2 respectively. No wave at cell(i, j) at time t means ***Zt_i,j = 0.0***. The water surface can go up and down between ***20.0*** and ***-20.0*** through the wave dissemination.



Schroedinger's wave formula computes ***Zt_i,j*** (where t >= 2) as follows:

$$Zt\_i,j = 2.0 * Zt\text{-}1\_i,j - Zt\text{-}2\_i,j + c^2 * (dt/dd)^2 * (Zt\text{-}1\_i+1,j + Zt\text{-}1\_i-1,j + Zt\text{-}1\_i,j+1 + Zt\text{-}1\_i,j-1 - 4.0 * Zt\text{-}1\_i,j)$$

where

c is the wave speed and should be set to 1.0,

dt is a time quantum for simulation, and should be set to 0.1, and

dd is a change of the surface, and should be set to 2.0.

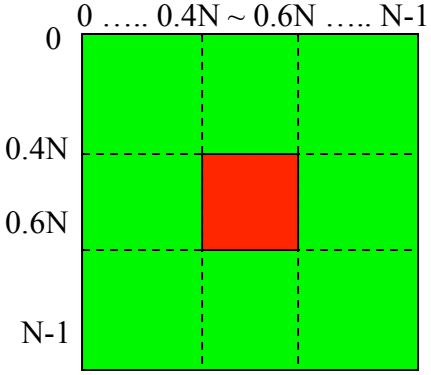
Note that, if a cell is on an edge, (i.e., i = 0, i = N -1, j = 0, or j = N – 1), ***Zt_i,j = 0.0***

The above formula does not work when t = 1. ***Zt_i,j*** (at t == 1) should be computed as:

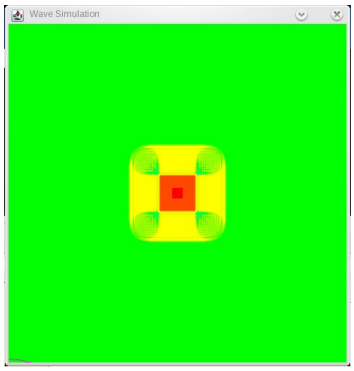
$$Zt\_i,j = Zt\text{-}1\_i,j + c^2 / 2 * (dt/dd)^2 * (Zt\text{-}1\_i+1,j + Zt\text{-}1\_i-1,j + Zt\text{-}1\_i,j+1 + Zt\text{-}1\_i,j-1 - 4.0 * Zt\text{-}1\_i,j)$$

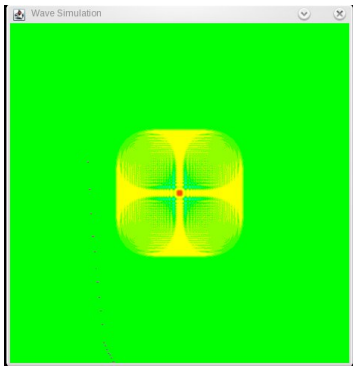
Note that, if a cell is on an edge, (i.e., i = 0, i = N -1, j = 0, or j = N – 1), ***Zt_i,j = 0.0***

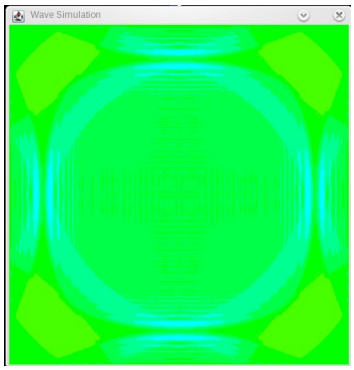
How about t == 0? This is an initialization of the water surface. Let's create a huge tidal wave in the middle of this square bucket. Set all cells(i, j) to 20.0 where 0.4 * N < i < 0.6 * N, 0.4 * N < j < 0.6 * N.



Your simulation now starts with t == 0 (initialization), increments t by one, and computes the surface height of all cells(i, j), (i.e., Zt_i,j) at each time t, based on the above formulae (See examples of simulation outputs below).







Look at ~css534/prog2/Wave2D_template.cpp. The main( ) function first reads three parameters: (1) size: the edge length of a 2D simulated square; (2) max_time: # steps to simulate wave dissemination where max_time >= 2; and (3) interval: # simulation steps needed each time the current simulation space is printed out, (e.g., interval == 1 means simulation status to be displayed every single step, interval == 2 prints out the space at time 2, 4, ..., whereas interval == 0 means no simulation output that is necessary to remove any I/O overheads when measuring execution performance.)

```
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;
  }
```

Thereafter, the main( ) function creates a simulation space and starts a timer.

```
  // 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( );
```

After that, main( ) initializes the water surface, z[0][][] at time = 0.

```
  // 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;
      }
    }
```

```
}
```

We now have to simulate the wave diffusion at time = 1, 2, and all the way to max_time - 1. You must implement the rest of main( ) by yourself. Don't forget to insert the code to print out the simulation space every interval steps. The printing format should be:

```
t
z[t][0][0] z[t][1][0] ... z[t][size-1][0]
z[t][0][1] z[t][1][1] ... z[t][size-1][1]
...
z[t][0][size-1] z[t][1][size-1] ... z[t][size-1][size-1]
```

For example, given z[3][3][3], when t == 2, we can print as follows:

```
2
z[2][0][0] z[2][1][0] z[2][2][0]
z[2][0][1] z[2][1][1] z[2][2][1]
z[2][0][2] z[t][1][2] z[2][2][2]
```

Note that, when t == 3, we cannot print z[3][i][j]. This in turn means that you have to shift down values from z[2][][] to z[1][][] and from z[1][][] to z[0][][]. Don't copy values, which slows down the execution. Instead, rotate z[2][][], z[1][][], and z[0][][].

Running this program is not really impressive unless its standard outputs are graphically displayed. For this purpose, use Wout.java. To see Wave2's outputs graphically, redirect the standard outputs to Wout. For example, to see a 100 x 100 simulation space every 10 steps from t = 0 to 499, type:

```
Wave2D 100 500 10 | java Wout 100
```

3. Parallelization

Follow the parallelization strategies described below:

(1) Copy Wave2D_template.cpp into Wave2D.cpp, and complete its sequential implementation. Check if it works correctly, using Wout.java.

(2) Copy Wave2D.cpp into Wave2D_mpi.cpp. Start parallelization with MPI first. Divide each square into small stripes along the i-axis. For instance, if you use four processes, z[3][100][100] should be divided and allocated to different processors as follows:

```
rank 0: z[0][0][j]  ~ z[0][24][j], z[1][0][j]  ~ z[1][24][j], z[2][0][j]  ~ z[1][24][j]
rank 1: z[0][25][j] ~ z[0][49][j], z[1][25][j] ~ z[1][49][j], z[2][25][j] ~ z[1][49][j]
rank 2: z[0][50][j] ~ z[0][74][j], z[1][50][j] ~ z[1][74][j], z[2][50][j] ~ z[1][74][j]
rank 3: z[0][75][j] ~ z[0][99][j], z[1][75][j] ~ z[1][99][j], z[2][75][j] ~ z[1][99][j]
```

Note 0 <= j < 99. For simplicity, each rank may allocate an entire z[3][size][size] array but just use only the above stripe.

(3) In each iteration of simulation loop t = 2 through to max_time - 1, your order of operations should be (a) printing out an intermediate simulation if necessary, (b) exchanging data among ranks, and (c) computing Schroedinger's formula in parallel.

(4) Rank 0 is responsible to print out an intermediate status to the standard output. For this purpose, rank 0 must receive all strips from the other ranks 1 ~ 3 before printing out the status.

(5) Two neighboring ranks must exchange boundary data. For instance, rank 1 must send its z[p][25][j] to rank 0 as well as z[p][49][j] to rank 2. At the same time, rank 1 must receive z[p][24][j] from rank 0 as well as z[p][50][j] from rank 2. At time == 2, p is 1. However, beyond time == 2, p will repeatedly change into 0, 1, and back to 2. Note that rank 0 has no left neighbor and rank N -1 has no right neighbor.

(6) Schroedinger's formula is the most computation intensive part that should be parallelized. Each rank computes only its own stripe.

(7) After verifying the correctness of your MPI parallelization, keep working on your parallelization with OpenMP. Focus on Schroedinger's formula.

4. Program Structure

Your work will start with completing Wave2D.cpp and thereafter parallelizing it into Wave2D_mpi.cpp. modifying Hea2D.cpp. Please login uw1-320-lab.uwb.edu and go to the ~css534/prog2/ directory. You will find the following files:

Program Name	Description
mpi_setup.txt	Instructs you how to set up an MPI environment on your account.
mpd.hosts	Lists four computing nodes used by MPI daemons. Choose four computing nodes among uw1-320-00 through to uw1-320-15, each having four CPU cores. In this way, we can evaluate our parallelization performance with up to 4 x 4 = 16 cores.
compile.sh	Is a shell script to compile all the professor's programs. Generally, all you have to do for compilation is: `mpic++ Wave2D_mpi.cpp Timer.cpp -fopenmp -o Wave2D_mpi`
Wave2D.cpp	Is the sequential version of the 2D wave-diffusion program. It is read -protected. You can't read it.
Wave2D_template.cpp	Is an incomplete version of Wave2D.cpp. This is what you have to complete first before parallelization.
Wave2D	Is the executable of Wave2D.cpp. To run the program, type: Wave2D size max_time interval Example: Wave2D 100 500 10 simulates wave diffusion over a 100 x 100 square for 500 cycles and prints an intermediate status every 10 cycles. If interval is 0, no output is printed out. To see the outputs graphically, you need to redirect the standard output to Wout.java. Example: Wave2D 100 500 10 | java Wout 100
Wout.java Wout.class	Is a Java program that reads Wave2D's ASCII outputs through a pipe and displays them graphically. This program needs to read the simulation size. In other words, Wave2D's first parameter and Wout's parameters must be the same value.
Wave2D_mpi.cpp	Is my key answer that parallelized WaveD.cpp. Needless to say, it is read-protected.
Wave2D_mpi	Is the executable code of MDmpi.cpp. mpirun –np #machines Wave2D_mpi size max_time interval #threads Example: mpirun –np 2 Heat2D_mpi 100 500 10 4 Use 2 machines, each spawning 4 threads to simulate wave diffusion over a 100 x 100 square for 500 cycles and prints an intermediate status every 10 cycles. If interval is 0, no output is printed out. For graphical outputs, type: mpirun –np 2 Heat2D_mpi 100 500 10 4 | java Wout 100 Note that you can't run it, because you have to run your own MPI program for performance evaluation.
Timer.h, Timer.cpp, Time.o	Is a program used in Wave2D.cpp and Wave2D_mpi.cpp to measure the execution time. Copy them in your working directory.

out1.txt and out4.txt	They are the outputs printed out by Wave2D and Wave2D_mpi (with 4 ranks) when simulating heat diffusion over a 100 x 100 square for 500 cycles. Therefore, they are identical.
measurements.txt	This file shows performance evaluation of the professor's Wave2D and Wave2D_mpi programs.

5. Statement of Work

Follow through the three steps described below:

Step 1: Complete Wave2D, parallelize Wave2D_mpi.cpp with MPI and OpenMP, and tune up its execution performance as much as you like.

Step 2: Verify the correctness of your Wave2D_mpi.cpp with your Wave2D.cpp as follows:

```
css534@uw1-320-10:~/prog2$ Wave2D 100 500 10 > out1.txt
Elapsed time = 391115
css534@uw1-320-10:~/prog2$ mpirun -n 4 Wave2D_mpi 100 500 10 4 > out4.txt
rank[0]'s range = 0 ~ 24
rank[1]'s range = 25 ~ 49
rank[2]'s range = 50 ~ 74
rank[3]'s range = 75 ~ 99
Elapsed time = 772250
css534@uw1-320-10:~/prog2$ diff out1.txt out4.txt
```

No difference should be detected between a sequential and a parallel execution.

Step 3: Conduct performance evaluation and write up your report. You should run your MPI/OpenMP version with the following scenarios:

```
mpirun -n x Wave2D_mpi 576 500 0 y
```

where X should be 1 through to 4 ranks and Y should be 1 through to 4 threads.

Below is the professor's program execution for the purpose of comparing your execution performance:

```
css534@uw1-320-10:~/prog2$ cat mpd.hosts
uw1-320-10.uwb.edu
uw1-320-11.uwb.edu
uw1-320-12.uwb.edu
uw1-320-13.uwb.edu
css534@uw1-320-10:~/prog2$ mpdboot -n 4 -v
running mpdallexit on uw1-320-10
LAUNCHED mpd on uw1-320-10  via
RUNNING: mpd on uw1-320-10
LAUNCHED mpd on uw1-320-11.uwb.edu  via  uw1-320-10
LAUNCHED mpd on uw1-320-12.uwb.edu  via  uw1-320-10
LAUNCHED mpd on uw1-320-13.uwb.edu  via  uw1-320-10
RUNNING: mpd on uw1-320-11.uwb.edu
RUNNING: mpd on uw1-320-12.uwb.edu
RUNNING: mpd on uw1-320-13.uwb.edu
css534@uw1-320-10:~/prog2$ Wave2D 100 500 10 > out1.txt
Elapsed time = 391115
css534@uw1-320-10:~/prog2$ mpirun -n 4 Wave2D_mpi 100 500 10 4 > out4.txt
rank[0]'s range = 0 ~ 24
rank[1]'s range = 25 ~ 49
rank[2]'s range = 50 ~ 74
rank[3]'s range = 75 ~ 99
Elapsed time = 772250
css534@uw1-320-10:~/prog2$ diff out1.txt out4.txt
css534@uw1-320-10:~/prog2$ Wave2D 576 500 0
Elapsed time = 2106527
css534@uw1-320-10:~/prog2$ mpirun -np 1 Wave2D_mpi 576 500 0 1
rank[0]'s range = 0 ~ 575
Elapsed time = 2101795
css534@uw1-320-10:~/prog2$ mpirun -np 1 Wave2D_mpi 576 500 0 2
rank[0]'s range = 0 ~ 575
Elapsed time = 1112091
css534@uw1-320-10:~/prog2$ mpirun -np 1 Wave2D_mpi 576 500 0 4
rank[0]'s range = 0 ~ 575
Elapsed time = 631505
css534@uw1-320-10:~/prog2$ mpirun -np 2 Wave2D_mpi 576 500 0 1
rank[1]'s range = 288 ~ 575
rank[0]'s range = 0 ~ 287
Elapsed time = 1335987
css534@uw1-320-10:~/prog2$ mpirun -np 2 Wave2D_mpi 576 500 0 2
```

```
rank[0]'s range = 0 ~ 287
rank[1]'s range = 288 ~ 575
Elapsed time = 893939
css534@uw1-320-10:~/prog2$ mpirun -np 2 Wave2D_mpi 576 500 0 4
rank[0]'s range = 0 ~ 287
rank[1]'s range = 288 ~ 575
Elapsed time = 575675
css534@uw1-320-10:~/prog2$ mpirun -np 4 Wave2D_mpi 576 500 0 1
rank[0]'s range = 0 ~ 143
rank[1]'s range = 144 ~ 287
rank[3]'s range = 432 ~ 575
rank[2]'s range = 288 ~ 431
Elapsed time = 931959
css534@uw1-320-10:~/prog2$ mpirun -np 4 Wave2D_mpi 576 500 0 2
rank[0]'s range = 0 ~ 143
rank[1]'s range = 144 ~ 287
rank[2]'s range = 288 ~ 431
rank[3]'s range = 432 ~ 575
Elapsed time = 681366
css534@uw1-320-10:~/prog2$ mpirun -np 4 Wave2D_mpi 576 500 0 4
rank[0]'s range = 0 ~ 143
rank[2]'s range = 288 ~ 431
rank[1]'s range = 144 ~ 287
rank[3]'s range = 432 ~ 575
Elapsed time = 525239
css534@uw1-320-10:~/prog2$ mpdallexit
css534@uw1-320-10:~/prog2$
```

Your minimum requirements to complete this assignment include:

(1) The performance improvement with four machines (i.e., four ranks) should be equal to or better than 2106527 / 931959 = 2.26 times

(2) The performance improvement with four machines (i.e., four ranks) with multithreading should be equal to or better than 2106527 / 525239 = 4.01 times

6. What to Turn in

This programming assignment is due at the beginning of class on the due date. Please upload the following materials to CollectIt. No email submission is accepted.

Criteria	Grade
Documentation of your parallelization strategies including explanations and illustration in one page.	20pts
Source code that adheres good modularization, coding style, and an appropriate amount of commends. • 25pts: well-organized and correct code receives • 23pts: messy yet working code or code with minor errors receives • 20pts: code with major bugs or incomplete code receives	25pts
Execution output that verifies the correctness of your implementation and demonstrates any improvement of your program's execution performance. • 25pts: Correct execution and better results than the two requirements: (1) four MPI ranks perform 2.2+ times better, and (2) four MPI ranks with multithreading perform 4.0+ times better than the sequential version. • 24pts: Correct execution and better results than requirement (1) four MPI ranks perform 2.2+ times better, but requirement (2) is just satisfied: four MPI ranks with multithreading perform 4.0+ times better than the sequential version. • 23pts: Correct execution and the two requirements just satisfied: (1) four MPI ranks perform 2.2 times better, and (2) four MPI ranks with multithreading perform 4.0 times better than the sequential version. • 22pts: Correct execution and requirement (1) was satisfied but requirement (2) was not satisfied.	25pts

• 20pts: Correct execution and better performance improvement but none of the two requirements satisfied. • 15pts: Correct execution but little performance improvement, (i.e., max. 1.3 times better or less) • 10pts: Wrong execution	
Discussions about the parallelization, the limitation, and possible performance improvement of your program in one page.	25pts
Lab Session 2 Please turn in your lab 2 by the due date of program 2. Your source code and execution outputs are required.	5pts
Total Note that program 2 takes 20% of your final grade.	100pts