CSS 543
Program 2: Multithreaded Schroedinger's Wave Simulation

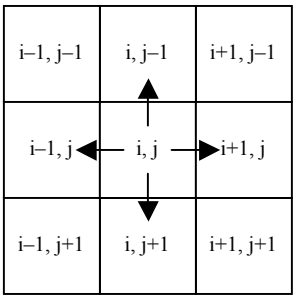
Professor: Munehiro Fukuda
Due date: see the syllabus

1. Purpose

This assignment uses multithreads to parallelize and thus to accelerate a simulation of Schroedinger's wave dissemination over a two-dimensional space. Through the parallelization, you will exercise inter-thread synchronization and tune up access patters to shared data among threads.

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:

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

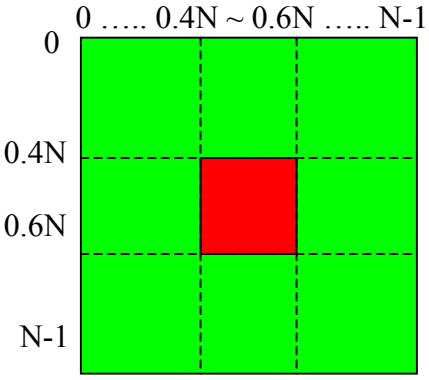
where
c is the wave speed and should be set to 1.0,
dt is a time quantum for simulation, and should be set to 0.1, and
dd is a change of the surface, and should be set to 2.0.
Note that, if a cell is on an edge, (i.e., i = 0, i = N -1, j = 0, or j = N – 1), ***Zt_i,j = 0.0***

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

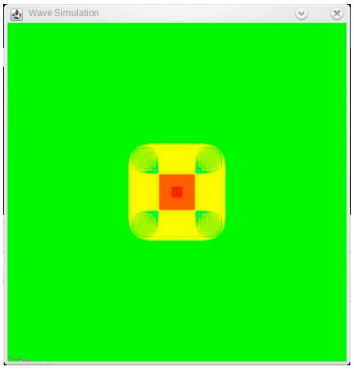
$$Zt\_i,j = Zt\text{-}1\_i,j + c^2 / 2 * (dt/dd)^2 * (Zt\text{-}1\_i+1,j + Zt\text{-}1\_i-1,j + Zt\text{-}1\_i,j+1 + Zt\text{-}1\_i,j-1 - 4.0 * Zt\text{-}1\_i,j)$$

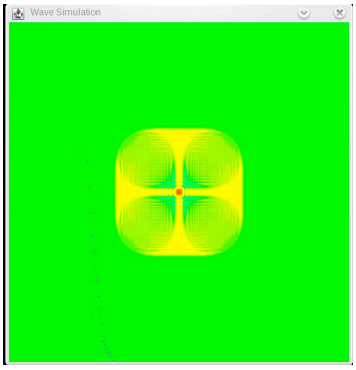
Note that, if a cell is on an edge, (i.e., i = 0, i = N -1, j = 0, or j = N – 1), ***Zt_i,j = 0.0***

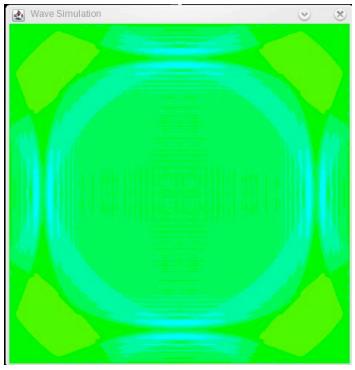
How about t == 0? This is an initialization of the water surface. Let's create a huge tidal wave in the middle of this square bucket. Set all cells(i, j) to 20.0 where 0.4 * N < i < 0.6 * N, 0.4 * N < j < 0.6 * N.



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).







3. Graphics

To observe an ongoing simulation, we need to graphically show the water surface in this square space at a given time t. Since the main purpose of this programming project is parallelization with multithreads, you do not have to spend too much time for graphics. Below, I will give you a framework of the program that gives a complete definition of methods regarding graphics. The code consists of five methods:

Methods	Descriptions
main( )	Instatiates double[3][N][N] in the reference "space". This is the array that maintains the surface height of all cells(i, j). space[2][i][j], space[1][i][j], and space[0][i][j] corresponds to ***Zt_i,j, Zt-1_i,j, and Zt-2_i.j***. The main( ) method passes this three-dimensional double array to Wave2D's constructor. You need to modify main( ) to receive more arguments than just the size of the simulation space, to spawn multiple threads, and to let them call Wave2D's run( ) method in parallel.
Wave2D( )	Is the Wave2D's constructor that substitutes space[3][N][N] in Z[][][], so that your simulation actually accesses Z[3][N][N]. At the very end, it calls startGraphics( ) to pop up a graphical window that shows the simulation space. You need to modify Wave2D( ) to handle more arguments than just the simulation space, Z[][][]. Let only one thread, namely the main thread call startGraphics( ) at the end of Wave2D( ).
startGraphics( )	Pops up a graphical window that shows the surface height of the simulation space. You don't have to modify it. Just use it as it is.
writeToGraphics( )	Updates the graphical window that shows the latest surface height of the simulation space. This function checks each Z[0][i][j]'s surface height, finds the corresponding color, and pastes the color to the corresponding cell of the graphical window. This method should be called from run( ) periodically in order to show the intermediate results to the monitor. You don't have to modify writeToGraphics( ). However, for better performance, you may even tune up this method. For instance, you can save the previous values of Z[0][i][j] and paste to the graphical window the color of only those cells whose value is different from the previous one.
run( )	Is the actual simulation that should be executed by each thread. The run( ) method should call writeToGraphics( ) periodically. You need to implement run( ), the heart of the simulation!

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Insets;
import javax.swing.JFrame;

class Wave2D implements Runnable {

    static final int defaultN = 100;  // the default system size
    static final int defaultCellWidth = 8;
    static final Color bgColor = new Color(255, 255, 255);//white background
    private int N = 0;                        // simulation size
    private double z[][][];                   // simulation space
    private JFrame gWin;                      // a graphics window
    private int cellWidth;                    // each cell's width in the window
    private Insets theInsets;                 // the insets of the window
    private Color wvColor[];                  // wave color

    public Wave2D( double[][][] space ) {
        this.z = space;
        this.N = z[0].length;
        startGraphics();
    }

    public void run( ) {
        // implement it!
    }

    private void startGraphics() {
        // the cell width in a window
        cellWidth = defaultCellWidth / (N / defaultN);
        if (cellWidth == 0) {
            cellWidth = 1;
        }

        // initialize window and graphics:
        gWin = new JFrame("Wave Simulation");
        gWin.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        gWin.setLocation(50, 50);  // screen coordinates of top left corner
        gWin.setResizable(false);
        gWin.setVisible(true);     // show it!
        theInsets = gWin.getInsets();
        gWin.setSize(N * cellWidth + theInsets.left + theInsets.right,
                     N * cellWidth + theInsets.top + theInsets.bottom);

        // wait for frame to get initialized
        long resumeTime = System.currentTimeMillis() + 1000;
        do {
        } while (System.currentTimeMillis() < resumeTime);

        Graphics g = gWin.getGraphics();
        g.setColor(bgColor);
        g.fillRect(theInsets.left,
                   theInsets.top,
                   N * cellWidth,
                   N * cellWidth);

        wvColor = new Color[21];
        wvColor[0] = new Color(0x0000FF);
        wvColor[1] = new Color(0x0033FF);
        wvColor[2] = new Color(0x0066FF);
        wvColor[3] = new Color(0x0099FF);
        wvColor[4] = new Color(0x00CCFF);
        wvColor[5] = new Color(0x00FFFF);
        wvColor[6] = new Color(0x00FFCC);
        wvColor[7] = new Color(0x00FF99);
        wvColor[8] = new Color(0x00FF66);
        wvColor[9] = new Color(0x00FF33);
        wvColor[10] = new Color(0x00FF00);
        wvColor[11] = new Color(0x33FF00);
        wvColor[12] = new Color(0x66FF00);
        wvColor[13] = new Color(0x99FF00);
        wvColor[14] = new Color(0xCCFF00);
        wvColor[15] = new Color(0xFFFF00);
        wvColor[16] = new Color(0xFFCC00);
        wvColor[17] = new Color(0xFF9900);
        wvColor[18] = new Color(0xFF6600);
```

```
        wvColor[19] = new Color(0xFF3300);
        wvColor[20] = new Color(0xFF0000);
    }

    private void writeToGraphics() {
        Graphics g = gWin.getGraphics();
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                // convert a wave height to a color index ( 0 through to 20 )
                int index = (int) (z[0][i][j] / 2 + 10);
                index = (index > 20) ? 20 : ((index < 0) ? 0 : index);

                g.setColor(wvColor[index]);
                g.fill3DRect(theInsets.left + i * cellWidth,
                             theInsets.top + j * cellWidth,
                             cellWidth, cellWidth, true);
            }
        }
    }

    public static void main(String args[]) {

        int N = Integer.parseInt( args[0] );

        // prepare a simulation space
        double[][][] space = new double[3][N][N];

        // instantiate slave threads
        Wave2D wave2D = new Wave2D( space );
    }
}
```

The above code is also available at **~css543/programming/project2/Wave2Frame.java**. You may copy it into your own working directory and modify it to complete your assignment.

4. Specification and Simulation Strategy

This simulation program must be code all in Wave2D.java. It should be invoked with the following three arguments:

java –Xmx512m Wave2D size time interval nThreads
where,
-Xmx512m: the memory size JVM can use.
size: the size of a square space. space[3][size][size] or N[3][size][size] will be created for simulation.
time: the logical time to stop your simulator. A simulation starts at 0 and completes at time.
interval: the interval time to update the graphical window. In other words, writeToGraphics( ) should be called every interval.
nThreads: the number of threads that should be involved in simulation. The main thread should be counted in nThreads.

Example:
Java –Xmx512m Wave2D 300 1000 50 4
Creates an N[3][300][300] simulation space, spawns three child threads, thus involve four threads (including the main thread), starts a simulation from time 0, updates the graphical window every 50 time units, (i.e, 0, 50, 100, 150, …), and ends the simulation at time 1000.

Assigned a different stripe of a simulation space, each thread should compute the water surface height of all cells within its own stripe at time t, thereafter synchronize with all the other threads (through a barrier synchronization), increment t by 1, and repeat this iteration until the end of the simulation.

Your program must also measure the execution time. Instantiate a Date object right before starting a simulation at time 0 (namely after creating a simulation space and spawning threads), instantiate a Date object again after all threads are synchronized upon the end of their simulation, and calculate the time difference between the second and the first Date objects.

Needless to say, your simulation program should run faster as you add more threads. Each uw1-320-16 through to uw1-320-23 machines have two dual-core CPUs, which in turn means that just up to four threads benefit the performance of your program. For performance evaluation with up to 8 threads, use "hercules.uwb.edu".

5. Statement of Work

Implement the multithreaded Wave2D.java simulation program as described above. For your program development, you may use any platform such as Unix, AIX, Linux, Mac OS, Open Solaris, and Windows as far as they support Java. Conduct verification and performance evaluation of your program on any of uw1-320-16 ~ uw1-320-22 machines. Measure the time elapsed to perform the following simulations:

```
java –Xmx512m Wave2D 500 2000 2000 1
java –Xmx512m Wave2D 500 2000 2000 2
java –Xmx512m Wave2D 500 2000 2000 4
```

After you have confirmed performance improvement with up to 4 CPU cores, repeat the same performance valuation with hercules.uwb.edu (or your own machine if yours has 8 CPU cores).

```
java –Xmx512m Wave2D 500 2000 2000 1
java –Xmx512m Wave2D 500 2000 2000 2
java –Xmx512m Wave2D 500 2000 2000 4
java –Xmx512m Wave2D 500 2000 2000 8
```

Note that the above test cases do not update graphics results at all. This is because we want to focus on only performance of multithreaded computation rather than include graphics overheads. For your functional verification, you should run Wave2D with various test cases that periodically update graphics results. Discuss about the performance improvement. You should also conduct additional evaluation by changing those parameters so as to explore more different behaviors of your program.

6. What to Turn in

This programming assignment is due at the beginning of class on the due date. Please turn in the following materials in a hard copy. No email submission is accepted.

Criteria	Grade
Documentation of your design such as thread allocation to the space, barrier synchronization, (and any modification onto startGraphics( ) and/or writeToGraphics( )) in one or two pages. Insufficient or too-much (i.e. less than 1 or more than 2 pages) documentation receives 4pts.	5pts
Source code that adheres good modularization, coding style, and an appropriate amount of comments. 5pts: well-organized and correct code receives 4pts: messy yet working code or code with a minor error 3pts: code with two errors/bug 2pts: code with 3+ errors/bugs or incomplete code.	5pts

Execution output that verifies the correctness of all your implementation as well as covers performance evaluation. 5pts: correct snapshots of your simulation and performance improvement with 8 CPUs 4pts: correct snapshots of your simulation and performance improvement with 4 CPUs 3pts: correct snapshots of your simulation and performance improvement with 2 CPUs 2pts: correct snapshots but no performance improvement 1pts: wrong snapshots or incomplete execution	5pts
Discussions about your performance evaluation and possible improvement scenario in one page. 5pts: Detailed performance evaluation with changing parameters to explore more different behaviors of your program. 4pts: Just discussions about what you had in the execution output 3pts: Insufficient discussions	5pts
Lab Sessions 3 through 4 counts 1pt for each. If you have not yet turned in a hard copy of your source code/output or missed any session(s), please turn in together with program 2.	2pts
Total Note that program 2 takes 15% of your final grade.	22pts