CSS 434:

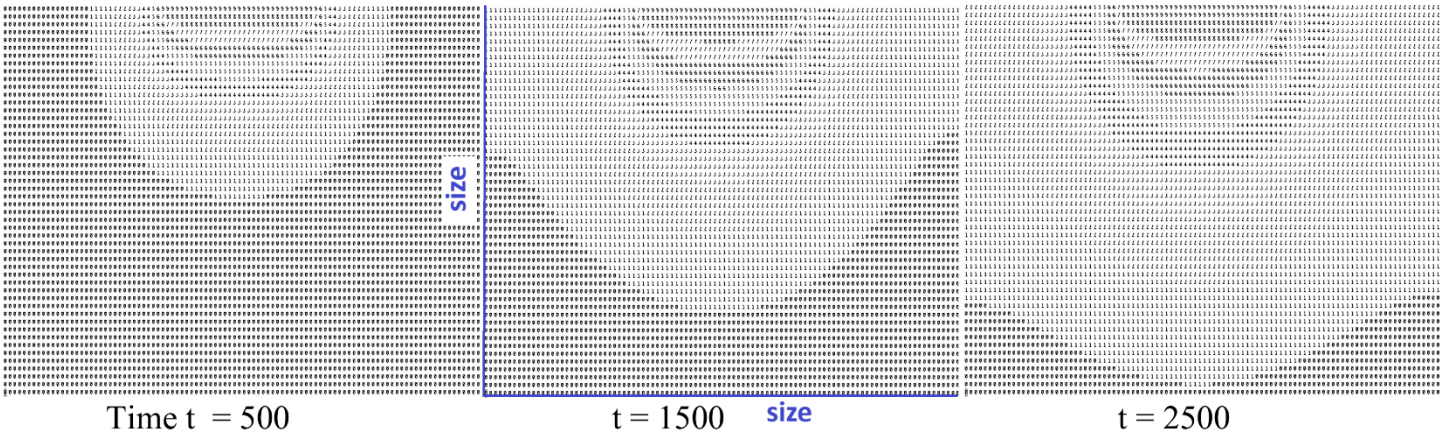
Homework 2:

Maryam M, October 2024.

Table of Contents

Summary

The program's objective is to simulate heat diffusion over a 2D $size * size$ square grid, where $size$ is the number of points per edge. Each point in the square is represented by an integer indicating the unit's temperature.



(Sample output provided in the assignment description and edited)

The program uses the Forward Euler method to simulate the heat diffusion using the finite difference method, where:

- $(x, y)_t$: The temperature of a point at time t.
  Depends on its own temperature, heat outflow, and heat inflow from its neighbors.
- r : Coefficient based on the diffusion rate, where $r = heat\ speed * \frac{time\ quantum}{(change\ in\ system)^2}$.

Inflow from neighbor points' previous temp. Heat Outflow

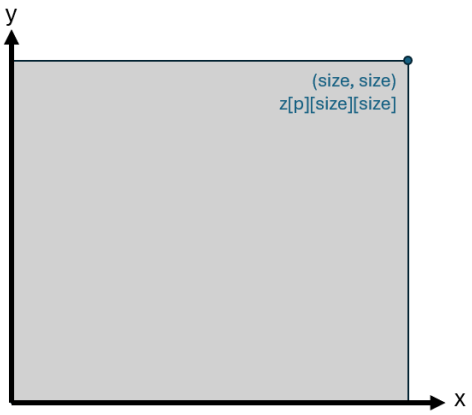
$$\underbrace{(x,y)_t}_{\text{current temp.}} = \overbrace{(x,y)_{t-1}}^{\text{prev. temp.}} + r((x+1,y)_{t-1} - (2*(x,y)_{t-1}) + (x-1,y)_{t-1}) + r((x,y+1)_{t-1} - (2*(x,y)_{t-1}) + (x,y-1)_{t-1})$$

RIGHT P. to right is left LEFT P.
TOP P. to top is bottom BOTTOM P.

(Formula used in heat diffusion program)

Non-Parallelized Version (Provided)

The non-parallelized version uses a 3D array $z[p][x][y]$ where x and y range from $0 - size$.



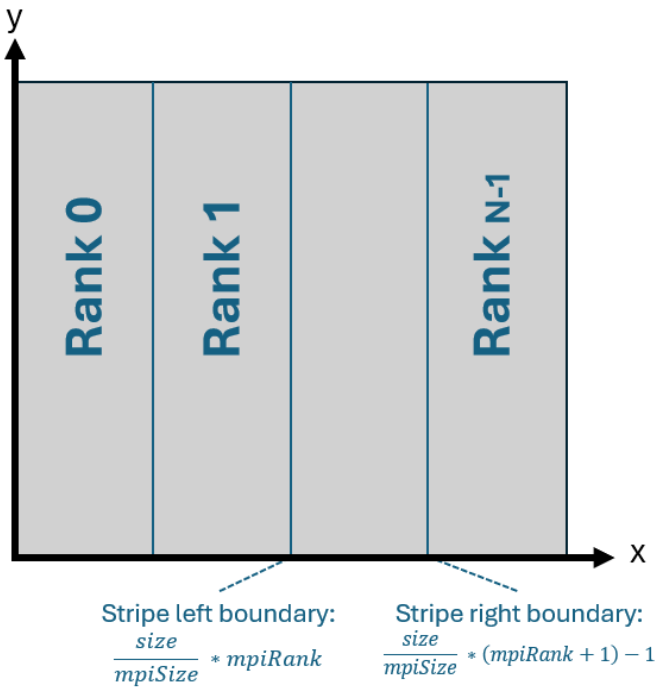
The grid is still 2D. p is a binary value used to track between previous and current temperatures in each cycle.

- When t is even, $p = 0$ indicates the previous heat status $(t - 1)$, And $p = 1$ indicates the current status we need to compute (t).
- When t is odd, $p = 1$ indicates the previous heat status $(t - 1)$, And $p = 1$ indicates the current status we need to compute (t).

$$(x, y)_{t-1} = z[\quad t \,\%\, 2 \quad][x][y]$$
$$(x, y)_{t} \quad = z[(t+1)\%\, 2][x][y]$$

Parallelized Version

The parallelized version follows the same logic, but splits up the grid into columns for each node.



Additionally, because we're not to use multi-dimensional arrays, we map values into a 1D array:

$$z[2][size][size] = \ z\_1D[2 * size * size]$$
$$z[p][\ \ x\ \ ][\ y\ \ ] \rightarrow z\_1D[p * size\ * \ size + x\ * \ size + y]$$

Before computing the stripes values, each rank receives it's neighbor's boundary so it can calculate it's own.

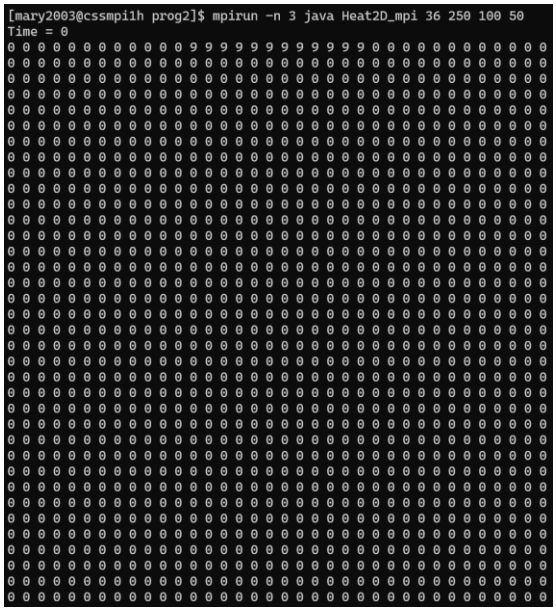
Execution Output

Using small values to check changer per iteration of *t*.

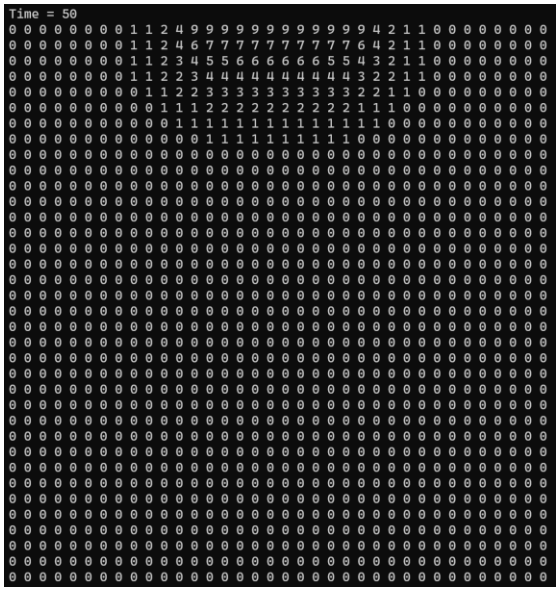
```
[mary2003@cssmpi1h prog2]$ mpirun -n 3 java Heat2D_mpi 12 8 8
Time = 0
0 0 0 0 9 9 9 9 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0

Time = 1
0 0 0 0 9 9 9 9 0 0 0 0
0 0 0 0 2 2 2 2 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0

Time = 2
0 0 0 0 9 9 9 9 0 0 0 0
0 0 0 0 2 3 3 2 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0

Time = 3
0 0 0 0 9 9 9 9 0 0 0 0
0 0 0 0 3 4 4 3 0 0 0 0
0 0 0 0 0 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
```

Large values for pattern view

Time = 0



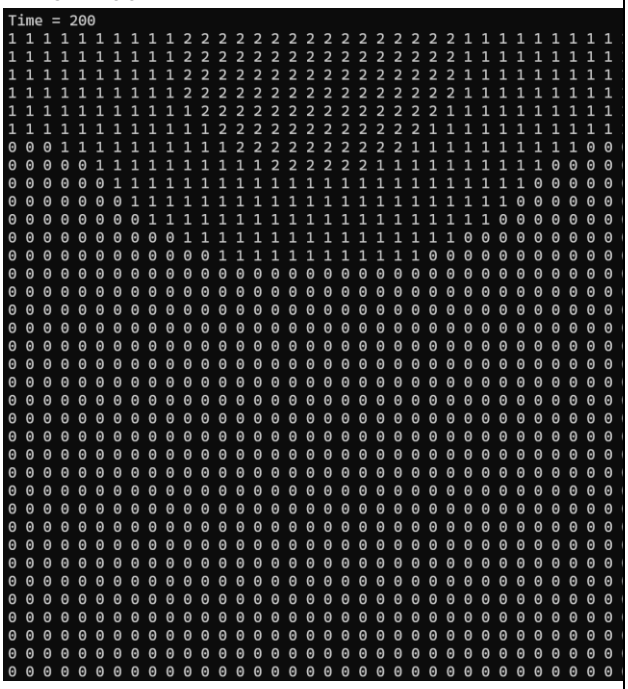
Time = 50



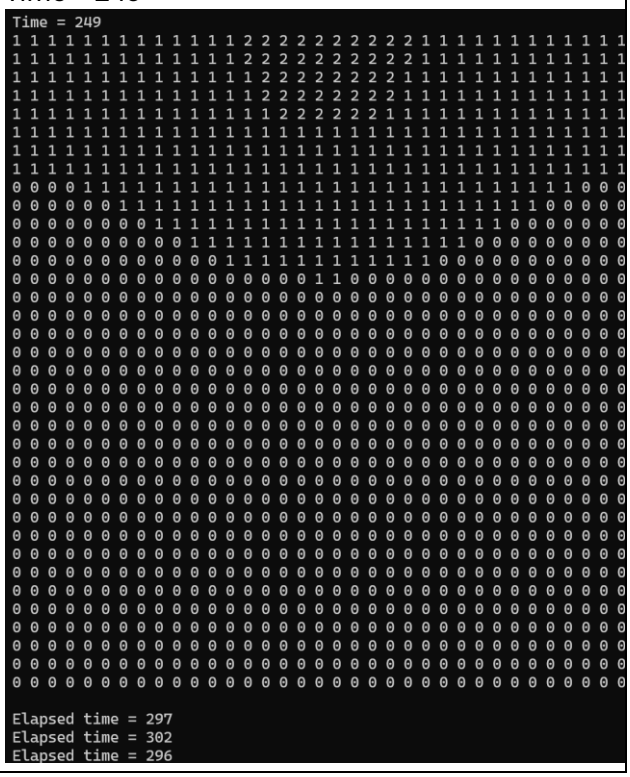
Times 50-199 skipped for space

```
Time = 4
0 0 0 1 9 9 9 9 1 0 0 0
0 0 0 1 3 4 4 3 1 0 0 0
0 0 0 0 1 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0

Time = 5
0 0 0 1 9 9 9 9 1 0 0 0
0 0 0 1 4 4 4 4 1 0 0 0
0 0 0 0 1 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0

Time = 6
0 0 0 1 9 9 9 9 1 0 0 0
0 0 0 1 4 5 5 4 1 0 0 0
0 0 0 0 1 2 2 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0

Time = 7
0 0 0 1 9 9 9 9 1 0 0 0
0 0 0 1 4 5 5 4 1 0 0 0
0 0 0 1 2 2 2 2 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0

Elapsed time = 379
Elapsed time = 380
Elapsed time = 396
```

Time = 200



Time = 249



Performance Differences for Different Node Counts

Note: We could decide to have only rank 0 print it's execution time, but for curiosity's sake we're keeping all the prints. Additionally, have the number of prints reflect on the number of nodes helps provide execution proofs.

Command parameters:

```
$ mpirun -n <node count> java Heat2D_mpi 36 250 150 50
```

Where node count = [1-4]

Node Count = 1	Node Count = 2	Node Count = 3	Node Count = 4
Elapsed time = 4003 [mary2003@cssmpi1h p	Elapsed time = 4924 Elapsed time = 4923 [mary2003@cssmpi1h prog	Elapsed time = 4833 Elapsed time = 4833 Elapsed time = 4801 [mary2003@cssmpi1h pi	Elapsed time = 4664 Elapsed time = 4667 Elapsed time = 4722 Elapsed time = 4662 [mary2003@cssmpi1h prog

Discussion

Parallelization

The program parallelizes the 2D heat diffusion simulation by dividing the computational domain into vertical stripes, each assigned to a different MPI process. Each process is responsible for updating heat values within its stripe using the Forward Euler method, which relies on temperature values from adjacent cells.

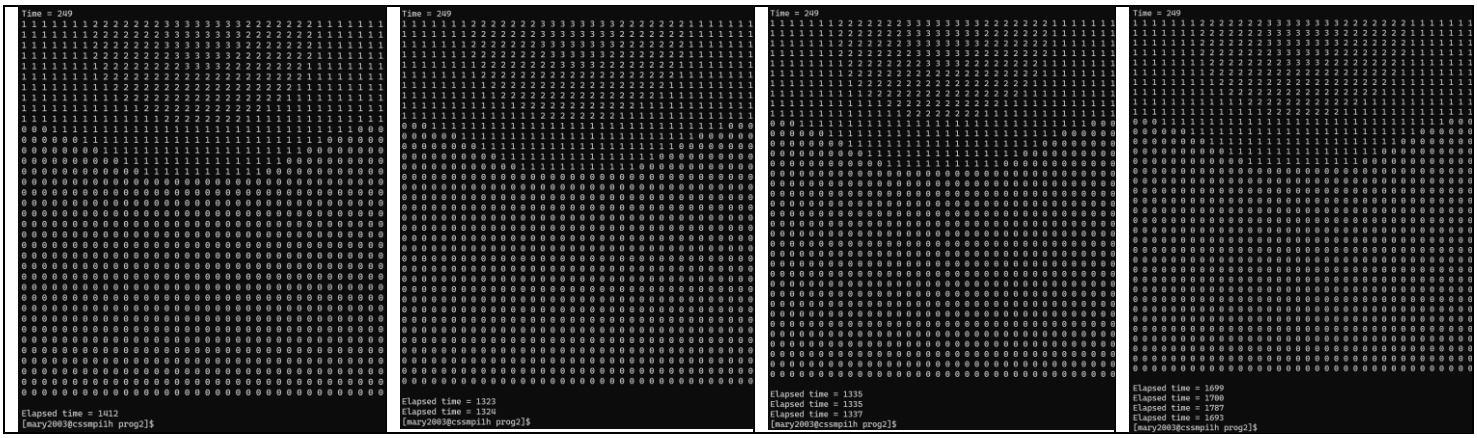
Note: To handle boundaries between stripes, each process exchanges boundary data with its neighbors at each time step, since each point relies on its neighbor's temperature to calculate its own.

This allows each process to simultaneously work on a smaller portion of the array, which should reduce the computation time.

Limitations

At first, there were a few unexpected results where less nodes meant less elapsed time.
With more tests, it became evident that the issue became worse the more timestamps there were. This possibly meant that the communication time for the lab nodes is expensive.

Node Count = 1	Node Count = 2	Node Count = 3	Node Count = 4
Elapsed time = 1412	Elapsed time = 1323 Elapsed time = 1324	Elapsed time = 1335 Elapsed time = 1334 Elapsed time = 1337	Elapsed time = 1699 Elapsed time = 1700 Elapsed time = 1787 Elapsed time = 1693



Possible Improvements

Generally, when the data set is smaller, using too many nodes is counter-productive because the communication costs outweigh the gains in computation costs.

On lowering the communication amount (smaller time variables), and larger windows (meaning each node does more work), results looks more as expected.

However the elapsed time for the same exact command could be hundreds off between tests, so using a more reliable system would improve test times.

Source Code

(Note: Head2D.java was provided, and thus not implemented in this assignment).

Heat2D_mpi.java

This is the parallelized version of Heat2D.java.

```
// Heat2D.java, edited to follow program 2 instructions.

import mpi.MPI;
import mpi.MPIException;
import java.util.Date;

public class Heat2D_mpi {
   private static double a = 1.0;  // heat speed
   private static double dt = 1.0; // time quantum
   private static double dd = 2.0; // change in system

   public static void main( String[] args ) {
      // Verify arguments.
      if ( args.length != 4 ) {
         System.out.
```

```
      println( "usage: " +
            "java Heat2D_mpi size max_time heat_time interval" );
   System.exit( -1 );
}

int size = Integer.parseInt( args[0] );
int max_time = Integer.parseInt( args[1] );
int heat_time = Integer.parseInt( args[2] );
int interval  = Integer.parseInt( args[3] );
double r = a * dt / ( dd * dd );

try {
   // Initialize MPI.
   MPI.Init(args);
   int mpi_size = MPI.COMM_WORLD.Size(); // Number of nodes.
   int mpi_rank = MPI.COMM_WORLD.Rank(); // ID of "this" node.

   // Each rank is only responsible for calculating it's stripe.
   int stripeLeftBoundary =  (size/mpi_size) * mpi_rank;
   int stripeRightBoundary = (size/mpi_size) * (mpi_rank + 1) - 1;
   int stripeSize = stripeRightBoundary - stripeLeftBoundary + 1;

   // Create a space in 1D array form:
   double[] z_1D = new double[2 * size * size];
   for ( int p = 0; p < 2; p++ ) {
      for ( int x = 0; x < size; x++ ) {
         // Two upper rows are identical
         z_1D[p*size * size+x * size+0] = z_1D[p*size * size+x * size+1];

         // Two lower rows are identical
         z_1D[p*size * size+x * size+(size-1)] = z_1D[p*size * size+x * size+(size-2)];

         for ( int y = 0; y < size; y++ ) {
            // Two leftmost columns are identical
            z_1D[p*size * size+0 * size+y] = z_1D[p*size * size+1 * size+y];

            // Two rightmost columns are identical
            z_1D[p*size * size+(size -1) * size+y]  = z_1D[p*size * size+(size-2) * size+y];
         }
      }
   }


   // Start a timer.
   Date startTime = new Date( );

   // Simulate heat diffusion:
   for (int t = 0; t < max_time; t++ ) {
```

```
int p = t % 2; // p = 0 or 1: indicates the phase
int p2 = (p + 1) % 2;

// First 3 loops are to be done simultanously (in each time step):

// Loop 1: Make two left-most and two right-most columns identical.
for ( int y = 0; y < size; y++ ) {
   z_1D[p*size * size+0 * size+y] = z_1D[p*size * size+1 * size+y];
   z_1D[p*size * size+(size-1) * size+y] = z_1D[p*size * size+(size-2) * size+y];
}

// Loop 2: Make two upper and lower rows identical.
for ( int x = 0; x < size; x++ ) {
   z_1D[p*size * size+x * size+0] = z_1D[p*size * size+x * size+1];
   z_1D[p*size * size+x * size+(size-1)] = z_1D[p*size * size+x * size+(size-2)];
}

// Loop 3: Keep heating the bottom until t < heat_time.
if ( t < heat_time ) {
   for ( int x = size /3; x < size / 3 * 2; x++ ) {
      z_1D[p*size * size+x * size+0] = 19.0; // Heat.
   }
}

// After 3 loops, exchange boundary data between neighboring ranks.
// Send and recieve to/from neighbors, unless boundary:

if (mpi_rank > 0) { // Handle Left neighbor.
   // Send the first column of this rank's stripe to the left neighbor
   int offset = p * size * size + (stripeLeftBoundary * size);
   MPI.COMM_WORLD.Send(z_1D, offset, size, MPI.DOUBLE, mpi_rank - 1, 0);

   // Receive the last column from the left neighbor and place it before the start of this rank's stripe
   offset = p * size * size + ((stripeLeftBoundary - 1) * size);
   MPI.COMM_WORLD.Recv(z_1D, offset, size, MPI.DOUBLE, mpi_rank - 1, 0);
}

if (mpi_rank + 1 < mpi_size) { // Handle Right neighbor.
   // Send the last column of this rank's stripe to the right neighbor
   int offset = p * size * size + (stripeRightBoundary * size);
   MPI.COMM_WORLD.Send(z_1D, offset, size, MPI.DOUBLE, mpi_rank + 1, 0);

   // Receive the first column from the right neighbor and place it after this rank's stripe
   offset = p * size * size + (stripeRightBoundary + 1) * size;
   MPI.COMM_WORLD.Recv(z_1D, offset, size, MPI.DOUBLE, mpi_rank + 1, 0);
}
```

```
        // On 4th loop, rank 0 collects data and prints status.
        if (mpi_rank != 0) {  // Ranks 1+ need to send their data to 0.
          int offset = p * size * size + (stripeLeftBoundary * size);
          MPI.COMM_WORLD.Send(z_1D, offset, stripeSize * size, MPI.DOUBLE, 0, 0);

        } else { // Rank 0:
          // Collect data from other ranks.
          for (int node = 1; node < mpi_size; node++) {
            int nodeLeftBound = stripeSize * node;
            int offset = p * size * size + (nodeLeftBound * size);

            // Receive the stripe from the node directly into the appropriate section of z_1D
            MPI.COMM_WORLD.Recv(z_1D, offset, stripeSize * size, MPI.DOUBLE, node, 0);
          }

          // Display intermediate results.
          if (interval != 0 && (t % interval == 0 || t == max_time - 1)) {
            System.out.println( "Time = " + t );

            for ( int y = 0; y < size; y++ ) {
              for ( int x = 0; x < size; x++ ) {
                System.out.print((int)(Math.floor(z_1D[p*size * size+x * size+y]/2) ) + " " );
              }

              System.out.println( );
            }
            System.out.println( );
          }
        }

        // On 5th loop, each rank computes stripe using forward Euler
        for (int x = stripeLeftBoundary; x <= stripeRightBoundary; x++ ) {
          if ((x == 0) || x == (size -1)) {
            continue;
          }
          for (int y = 1; y < size - 1; y++) {
              z_1D[p2*size * size+x * size+y] = z_1D[p*size * size+x * size+y] +
                r * (z_1D[p*size * size+(x+1) * size+y] - 2 * z_1D[p*size * size+x * size+y] + z_1D[p*size * size+(x-1) *
size+y]) +
                r * (z_1D[p*size * size+x * size+(y+1)] - 2 * z_1D[p*size * size+x * size+y] + z_1D[p*size * size+x *
size+(y-1)]);
          }
        }

      } // End of simulation

      // Finalize MPI
      MPI.Finalize();
```

```
        // Finish the timer
        Date endTime = new Date( );
        System.out.println( "Elapsed time = " + (endTime.getTime( ) - startTime.getTime( )));

      } catch (MPIException mpiExc) {
        mpiExc.printStackTrace();

      }

    }
}
```