CSS 434
Program 2: MPI Java Application

Professor: Munehiro Fukuda
Due date: see the syllabus

1. Purpose

In this programming assignment, we will use MPI Java to parallelize a sequential version of a two-dimensional heat diffusion program.

2. MPI Java

MPI: Message Passing Interface is the most well known standard used for coding parallel-computing applications, based on the message-passing programming paradigm. MPI-CH and Open-MPI are actual libraries that have implemented MPI functions in C and C++. However, Java programs cannot use these libraries directly, for which reason mpiJava was designed to bridge Java applications to the underlying MPI-CH functions.

Our Linux environment in cssmpi1-8.uwb.edu has already installed MPI-CH and mpiJava. Your Java application is executed on top of JVM, and mpiJava functions called in your program will use the underlying MPI-CH functions that exchange various messages among different computing nodes.

First of all, you need to set up you shell environment so as to run the latest version of MPI-CH. Login any cssmpi1-8.uwb.edu machine and follow the instructions in the home/mfukuda/css434/lab2a/mpi_setup.txt file.

3. How to Code and Run an MPI Java Program

You can find MatrixMult.java in the /home/mfukuda/css434/lab2a/ directory. This is an mpiJava application that computes matrix multiplication. As in the code, all mpiJava applications must import the following package:

```
import mpi.*;
```

To start and stop mpiJava, the main(String[] args) function must call the following functions at the beginning and the end respectively:

```
MPI.Init( args );
MPI.Finalize( );
```

Between these function calls, you may use any mpiJava functions such as:

`MPI.COMM_WORLD.Bcast( );` Broadcast an array to all processes.
`MPI.COMM_WORLD.Send( );` Send an array to a destination process.
`MPI.COMM_WORLD.Recv( );` Receive an array sent from a source process.

In mpiJava, each process engaged in the same computation is identified with a rank (starting from 0.) You may use the rank 0 process as a master and the other processes as a worker. You can find each process' rank information and the number of processes engaged in the computation through:

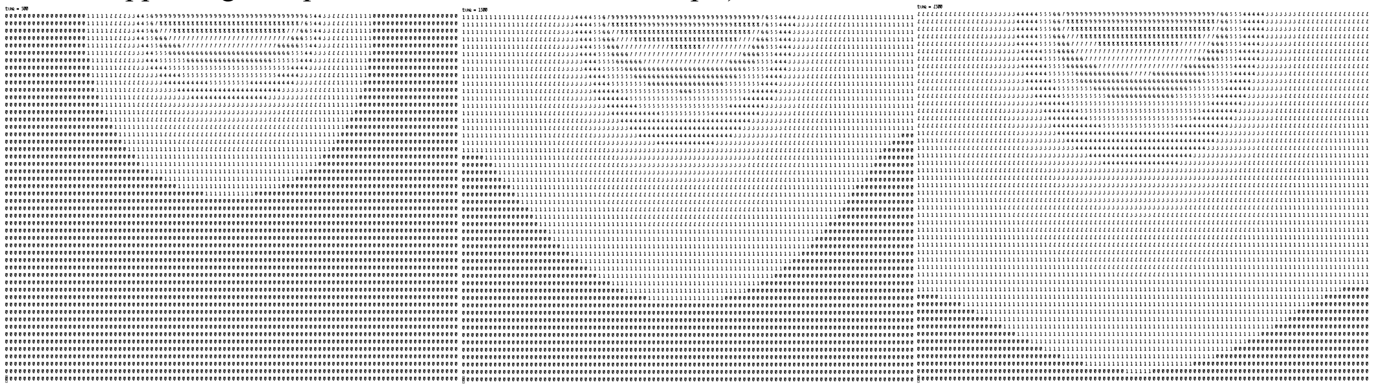
```
MPI.COMM_WORLD.Rank( );
MPI.COMM_WORLD.Size( );
```

All messages handled in mpiJava must be an array. Don't use multi-dimensional arrays. You cannot send a primitive data like int, float, and double. If you want to send one int, float, or double value, you have to first allocate an array of int[1], float[1], or double[1], and thereafter pass its reference to an mpiJava functions.

For more details, look at the professor's example code and read the mpiJava documentation:
Tutorial: http://www.hpjava.org/courses/arl/lectures/mpi.ppt
Specification: http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec.pdf

4. Two-Dimensional Heat Diffusion Program

This program simulates heat diffusion over a square space. Heated one third of the upper edge of 100 x 100 square space, the program shows the downward diffusion of the heat for the next 3000 steps. (Note that the upper edge keeps heated for the first 2700 steps.)



Time t = 500 t = 1500 t = 2500

The main( ) function creates a z[2][size][size] array where size is the edge length of a simulated square, (e.g. 100 in the above pictures and 1000 in our performance evaluation). The reason why we need two squares such as z[0][size][size] and z[1][size][size] is that, at a given simulation time t, if t is an even number (0, 2, 4, …), we use z[0][size][size] as the current status of heat diffusion and z[1][size][size] as the next status we need to compute, whereas if t is an odd number (1, 3, 5, …), we use z[1][size][size] as the current status and z[0][size][size] as the next status. For this purpose, main( ) uses the variable p = (t % 2) to indicate which square to use as the current status and the variable p2 = (p + 1) % 2 to indicate the other square.

After this space creation, main( ) iterates the following five loops at each simulation time t:
(1) Make two leftmost and two rightmost columns identical.

```
for ( int y = 0; y < size; y++ ) {
  z[p][0][y] = z[p][1][y];               // two leftmost columns are identical
  z[p][size - 1][y] = z[p][size - 2][y]; // two rightmost columns are identical
}
```

(2) Make two upper and lower rows identical.

```
for ( int x = 0; x < size; x++ ) {
  z[p][x][0] = z[p][x][1];               // two upper rows are identical
  z[p][x][size - 1] = z[p][x][size - 2]; // two lower rows are identical
}
```

(3) Keep heating the middle one third of the upper row until t < heat_time, (i.e., 2700 in our simulation). The value of heat ranges 0.0 through to 19.0.

```
if ( t < heat_time ) {
  for ( int x = size /3; x < size / 3 * 2; x++ )
    z[p][x][0] = 19.0;                   // heat
}
```

(4) Display the current status every interval simulation time. Note that, if interval == 0 for evaluating execution performance, the program does not show the status at all. The heat value will be converted into a character as follows:

z[p][x][y] value	Printed character
0.00 ~ 2.00	0
2.01 ~ 4.00	1
4.01 ~ 6.00	2
…	…
18.01 ~ 20.00	9

```
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[p][x][y] / 2 ) + " "  );
    System.out.println( );
  }
  System.out.println( );
}
```

(5) Compute the next status, using the Forward Euler method. Each array element needs values from its four neighbors (in east, west, north, and south) to update itself.

```
int p2 = (p + 1) % 2;
for ( int x = 1; x < size - 1; x++ )
  for ( int y = 1; y < size - 1; y++ )
    z[p2][x][y] =
      z[p][x][y] +
      r * ( z[p][x + 1][y] - 2 * z[p][x][y] + z[p][x - 1][y] ) +
      r * ( z[p][x][y + 1] - 2 * z[p][x][y] + z[p][x][y - 1] ) ;
```

5. Parallelization

Follow the parallelization strategies described below:

(1) Copy Head2D.java into Head2D_mpi.hava. Divide the 2D simulation space into small stripes along the x-axis. For instance, if you use four processes, z[2][100][100] should be divided and allocated to different processors as follows:

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

Note 0 <= y < 99. For simplicity, each rank may allocate an entire z[2][size][size] array but just use only the above stripe.

(2) For the five loops explained above, the first three loops are too small to parallelize. Therefore, all ranks can execute these three loops simultaneously.

(3) After these three loops, you will have to exchange boundary data between two neighboring ranks. For instance, rank 1 must send its z[p][25][y] to rank 0 as well as z[p][49][y] to rank 2. At the same time, rank 1 must receive z[p][24][y] from rank 0 as well as z[p][50][y] from rank 2. Note that rank 0 has no left neighbor and rank N -1 has no right neighbor.

(4) The 4th loop prints out an intermediate status to the standard output. Only rank 0 is responsible to print it out. For this purpose, rank 0 must receive all stripes from the other ranks 1 ~ 3 before printing out the status.

(5) The 5th loop is the most computation intensive part that should be parallelized. Each rank computes its own stripe, using the Forward Euler method.

(6) z[2][size][size] must be converted in a single-dimensional array, z[2 * size * size]. Any reference to z[p][x][y] must be converted in z[p * size * size + x * size + y].

6. Program Structure

Your work will start with modifying Hea2D.cpp that the professor got prepared for. Please login cssmpi1-8.uwb.edu and go to the /home/mfukuda/css434/hw2/ directory. You can 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 three remote computing nodes (in addition to your current local machine) used by MPI daemons. Choose three remote computing nodes among cssmpi1 through to cssmpi8, each having three CPU cores.
Heat2D.java	Is the sequential version of the 2D heat diffusion program that you will parallelize.
Heat2D.class	Is the compiled class file of Head2D.java. To run the program, type: Head2D size max_time heat_time interval Example: Heat2D 100 3000 2700 500 simulates heat diffusion over a 100 x 100 square for 3000 cycles and prints an intermediate status every 500 cycles. If interval is 0, no output is printed out. Note that the upper edge of the square will be heated for the first 2700 simulation cycles.
Heat2D_mpi.java	Is my key answer that parallelized Heat2D.java. Needless to say, it is read-protected.
Heat2D_mpi	Is the compiled class file of Heat2D_mpi.java. prunjava #machines Head2D_mpi size max_time heat_time interval Example: prunjava 2 Heat2D_mpi 100 3000 2700 500 uses 4 machines to simulate heat diffusion over a 100 x 100 square for 3000 cycles and prints an intermediate status every 500 cycles. If interval is 0, no output is printed out.
out1.txt and out4.txt	They are the outputs printed out by Heat2D and Heat2D_mpi (with 4 ranks) when simulating heat diffusion over a 100 x 100 square for 3000 cycles. Therefore, they are identical.
measurements.txt	This file shows performance evaluation of the professor's Heat2D and Heat2D_mpi programs.

7. Statement of Work

Follow through the three steps described below:

Step 1: Parallelize Heat2D.java with MPI Java, and tune up its execution performance as much as you like.

Step 2: Verify the correctness of your Heat2D_mpi.cpp with Heat2D.cpp as follows:

```
[mfukuda@cssmpi1 hw2]$ java Heat2D 100 3000 2700 500 > out1.txt
[mfukuda@cssmpi1 hw2]$ java_mpirun 4 Heat2D_mpi 100 3000 2700 500 > out4.txt
[mfukuda@cssmpi1 hw2]$ nano out4.txt
Remove all MPI-generated messages just before time = 0. Also remove the last two lines that
include "Elapsed time = …".
[mfukuda@cssmpi1 hw2]$ 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 and measure the execution performance of your Heat2D_mpi:

```
java_mpirun x Heat2D_mpi 1000 3000 2700 0
```

where X should be 1 through to 4 machines.

8. 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 soft copy to Canvas.

Criteria	**Grade**
Documentation of your parallelization strategies including explanations and illustration in one page.	5pts
Source code that adheres good modularization, coding style, and an appropriate amount of commends. (It may be included in your PDF/Word report or submitted as independent files.) (1) A conversion from a 3D array to a 1D array: 1pt (if incorrect, 0.5pts) (2) An implementation of exchange boundary: 1pt (if incorrect, 0.5pts) (3) A display of intermediate results: 1pt (if incorrect, 0.5pts) (4) A parallel execution of the forward Euler method: 1pt (if incorrect, 0.5pts) (5) Code completeness: 0.5pts (6) Coding style and readability (0.5pts)	5pts
Execution output that verifies the correctness of your implementation and demonstrates any improvement of your program's execution performance. (1) 5pts: Correct execution and performance improvement from 1 to 4 machines. (2) 4pts: Correct execution and performance improvement from 1 to 2 or 3 machines but not 4 machines. (3) 3pts: Correct execution but little performance improvement with 2, 3, or 4 machines. (4) 2pts: Wrong execution regardless of any performance improvement.	5pts
Discussions about the parallelization, the limitation, and possible performance improvement of your program in one page. (1) Current limitations and performance problems (2.5pts) (2) Possible performance improvement (2.5pts)	5pts
Total	20pts

Your lab2a and lab2b will be graded together with program 2.

Lab2a:

Outputs	1pt

Lab2b:

Source code	0.5pts
Outputs	0.5pts