RMRSUV002 REPORT A2

Introduction

The objective of the project is to design a multithreaded java program that simulates the flow of water across a terrain as it accumulates in basins i.e. local minimums in the terrain grid and flows off the edge of the terrain i.e. boundary points. For multithreaded programs to work well, it is essential to ensure thread safety and efficient concurrency. Concurrency entails correctly and efficiently managing access to shared resources from multiple clients. The program should be designed with thread safety and maximal concurrency in mind to ensure that the program displays liveliness, responsiveness and no deadlock. It should also conform to the Model View Controller pattern for user interfaces. The report will identify and explain how the "Water Flow Simulator" has made use of Java concurrency features, additions to the Skeleton code and a description of the user interface.

Objective

The goal of the project is to create a Water Simulator that runs on multiple threads and produces correct and efficient output. The expected behavior is that water will flow downhill, accumulating in basins and flow off the edge of the terrain gray scaled image which models the Terrain height two-dimensional array. The white colorations on the image indicate a high point of elevation whereas the black colorations indicate low points of elevation .It is hypothesized that the water should flow from high points of elevation towards low points of elevation resulting in the blocks of water dispersing towards the black portions of the image. The user should be able to click on the landscape image and add blocks of water represented by a blue coloured block appearing on click. This can take place in any state of the simulation i.e. whilst running or paused. The state of the simulation should be controlled by the user with the use of buttons. The GUI should update in real time displaying the behavior of the simulation accordingly.

Simulation approach

The methodology used for my implementation was to utilize one buffer image that being the terrain buffered image of the Terrain class. When a user clicks on the terrain buffered image the pixels of the image are altered to show a blue colour, indicating the presence of water resulting in water depths being recorded in the relevant two-dimensional water array. Once water flows away from a spot on the image the colour shading of the spot is recalculated- methodology was abstracted from Terrain classes deriveImage( ) method. A call to repaint shows the latest updated image in the content pane. The java Forkjoin framework was used due to performance reasons (find motivation below).This approach was used to implement a map algorithm to perform a large transformation on the image's pixels. It uses a divide and conquer approach making it superior to alternately running the simulation on just four threads.

Motivation for ForkJoin Framework as opposed to Java Threads as per an option of choice given by Professor Gain

Link to forum post

<https://vula.uct.ac.za/portal/site/57872ff9-5c75-4def-b501-80c784752dcf/tool/c33edf7b-fca0-43a4-aa9b-b4df04a8efbc/discussionForum/message/dfViewThread>

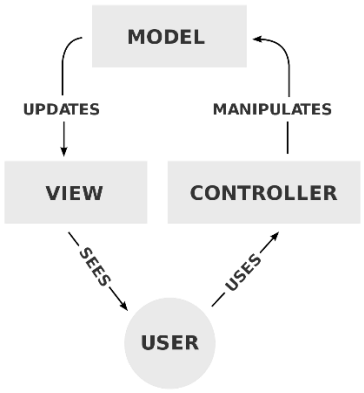
The assignment specification suggested that 4 threads were to be used to carry out the simulation of moving water around the terrain image with each thread operating on an equal portion of the Terrain landscape. This methodology allowed for easier synchronization however I noticed that this had a tradeoff ,that being, as the terrain size increased slower simulation runtimes occurred. I implemented an approach that used multiple threads to handle smaller sections of the two-dimensional grid, by means of using a Map Algorithm and the ForkJoin Framework. Recursive action was used to perform a large transformation on parts of the image's pixels thus moving the water across the landscape.

The implementation of the ForkJoin framework exhibited a much more responsive behavior when it came to problems of larger datasets since Java Threads are heavyweight and require a lot of overheads whereas the ForkJoin Threads are lightweight. It takes a lot of time switching between heavyweight threads as opposed to lightweight threads making parallel overheads dominate the simulations run time and drastically decrease performance when using Java threads.

The use of four threads on the large dataset lead to extremely slow simulation runtimes hence I chose to implement the ForkJoin framework as opposed to the four java threads due to its preferred efficiency.

Code Architecture

The Model View Controller pattern was used in this assignment



Model

Terrain.java

Class that stores information about terrain heights for each position in a two-dimensional array of type float.

Water.java
Class that stores information about the water depth at each position for the created water in a two-dimensional array of type int.

View
Flow.java
The flow class provided the GUI which the user will interact with in order to manipulate the Water Flow Simulator. The user can enter inputs through mouse clicks for adding water blocks or they can alter the state of the simulation using the buttons added to the Flow class.

Controller
FlowPanel.java
Refers to the main controller class responsible mainly for refreshing the GUI view after a transformation of the buffered image by the ForkJoin threads occur.

Parallel.java
This class extends Recursive Action class of the ForkJoin Framework and is responsible for the transformation of the land image. This algorithm implements a map and allows for more efficient updates to the terrain image as the problem scales.

Explanation of classes and alterations made to Skeleton code

Flow-Alterations

setupGUI

This method is a public static void method which takes three parameters, namely frameX, frameY which is of type integer and landdata which is of type Terrain. The setup GUI is responsible for the creation of the GUI. It is the main driver behind the Simulators visuals.

Alterations to the method:

- **Mouse Adapter**
  The mouse adapter was added to the main content panel "g" which is responsible for attaining the coordinates of the point on the image that was clicked. The mouse listener checks if the point clicked is not on the boundaries as water should not be added to the boundaries. If the clicked points meet the criteria, then a method "paintPixel( )" contained in the FlowPanel is called to alter the pixel color of surrounding points to blue, and record the addition of water in the data structure of choice.
- **Addition of components to the frame**
  Jlabel was added to display the current simulation step counter. The current simulation step count is determined by an Atomic integer value i.e. "atomicCounter". The text is updated using the increment and get method from Atomics. The atomic integer was chosen as it is safe to use in a multithreaded context.

  Three additional buttons were added to the frame that being play, pause and reset.

  Play button Action Listener calls a method to change the volatile Boolean flag "boolPause" to false which allows for the simulation to begin.

  Pause button Action Listener calls a method to change the volatile Boolean flag "boolPause" to true which pauses the simulation in its current state

  Reset button Action Listener is responsible for stopping the current simulation, removing the blocks of water that were added to the Terrain image and calling the necessary methods to reset the Simulation step counter and initialize all water values to 0.

Main( )

The method readWaterArray( ) is called under the Flow classes main method which initializes a Water Array with the same dimensions of the Terrain grid. Values are initialized to zero as no water is added to the landscape yet.

Terrain class

There were no alterations made to the Terrain class. It was used to model the landscape and manipulate a two-dimensional array that resembles the elevations of the landscape.

Water class

The water class is used to model the water that is added onto the terrain. It has a two-dimensional grid used to store water depths. This array has the exact same dimensions as the terrain grid. Thus, a height value in the two-dimensional Height array has a corresponding water depth at the same index.

A constant is declared for units of water added on click. For this project a value of 3 was chosen for the units of water added on click. It was declared as a constant, so that it can easily be changed, without needing to go through the entire code.

Methods

readWaterArray(Terrain terrain)

The method is a static void method that takes in a Terrain object as a parameter. It initializes an integer two-dimensional array with values of zero as no water is present on the initialization of the array. The bounds of the array are determined using the dimx and dimy from the Terrain object.

Synchronized Methods

The following methods have the keyword synchronized in the method definition. This is done to ensure thread safety as many threads read and write in the same resource. This prevents simultaneous access to a shared resource.

increaseWaterDepth(int x, int y)

This method is a static synchronized void method which takes in two parameters x and y which represents coordinates in the water two-dimensional grid. This method increments the current water value by 1.A value is incremented when water flows to the xy coordinate.

decreaseWaterDepth(int x, int y)

This method is a static synchronized void method which takes in two parameters x and y which represents coordinates in the water two-dimensional grid. This method decrements the current water value by 1. A value is decremented by a single unit when water flows away from the xy coordinate.

getSurafce(int x, int y)

This method is a static synchronized void method which takes in two parameters x and y which represents coordinates in the water two-dimensional grid. The getSurface method is responsible for converting water units to mm and adding the corresponding height value in the Terrain two-dimensional array

$$Surface_{xy} = WaterDepth_{xy} + TerrainHeight_{xy}$$

$$S_{xy} = W_{xy} + H_{xy}$$

FlowPanel

Volatile Boolean flag “boolPause” was used to start and pause the simulation. It was manipulated with the use of pauseSim( ) and playSim( ) methods.

Motivation behind the use of volatile Boolean as a flag:

Variables declared volatile are not cached and a read always returns the most recent write by any thread.

paintPixel(int x, int y)

This method is a public void that receives two integer variables x and y which represent the images pixel coordinates that were clicked. The method then contains two nested for loops which allows for painting surrounding pixels. Before a point is painted blue and depth is increased, a check whether the point is not a boundary point is performed. This check ensures that no water is added to the edge pieces. A call to the repaint is made to update the image that is being displayed.

moveWater(Terrain land, int xnew, int ynew, int xold, int yold)

The method moveWater is a public static void method which has 5 parameters.

The xnew and ynew coordinates resemble the points that are required to be updated with a new depth value and shade as water has flowed to this point.

The xold and yold coordinates resemble the points from where a unit of water has flown away from.These points require their corresponding depth value to be decremented and a new shading to be assigned.

The shading of the images points is determined using if statements

- Depth value of zero – This value indicates that the point no longer contains water implying that the color at that coordinate needs to be reverted to its original grayscale color. The color of the point is determined using the calculation below, which was abstracted from the terrain class.

```
float val = (land.height[xold][yold] - land.minh) / (land.maxh - land.minh);
Color col = new Color(val, val, val, 1.0f);
land.img.setRGB(xold, yold, col.getRGB());
```

- Depth values of one gets a light blue color assigned to it.
- Depth values of two gets a blue color assigned to it.
- Depth values greater than or equal to three gets assigned a dark blue color.

A call to repaint( ) shows the latest updated image.

Run( )

The run method is the main driver behind the simulation. When the boolPause variable changes state to false, it will allow for the fork join pool to be invoked and operate on the main task. The fork join operates on the permuted list of coordinates breaking the task into smaller subtasks that are easier to compute. My implementation of moving the water around follows a Divide and Conquer approach. Once the forkjoin has completed traversing the entire grid, the simulation counter is updated with the value of the atomic Integer “atomicCounter” with the use of the increment and get method. A call to repaint is made to view the updated terrain image. The simulation will continue until the boolPause variable evaluates to true. This is done by the Pause button

Parallel Class

The Parallel class extends Recursive action therefore it is a subclass of Recursive Action and it overrides the compute method. The problem in the assignment requires a transformation of the Terrain image, which makes Recursive Action better suited as opposed to Recursive Task as there is no final answer to return. The transformation in this instance is the moving of the water blocks across the terrain.

Constructor

The parallel class contains 5 instance variables namely land, height, water, lo, hi. These variables are instantiated in the constructor.

compute( )

This method is a protected void method which operates on the ArrayList permute "permute". The parameters lo and hi specify the range of elements in the to be processed. This helps splitting the task into sub tasks-if its size is greater than the specified *SEQUENTIAL_Cutoff*, otherwise the computation is performed on the whole list directly.

If the difference between hi and lo are greater than sequential cutoff, we divide the list into 2 parts and create two subtasks that process each. In turn, the subtask may be also divided further into smaller subtasks recursively until the size is less than the threshold, which invokes the computeDirectly() method.

computeDirectly()

This method synchronizes on the water object to ensure thread safety. When the difference between the upper bound hi and lower bound lo is less than the specified *SEQUENTIAL_Cutoff* the method then retrieves the index that it will operate on using the locate and get permute methods from the terrain class. These methods load an array with points to traverse. The permute methods allow for a more natural flow of water. Once the two-dimensional coordinates are located, the current surface is obtained for that position in the two-dimensional grid, this is done by making a call to the synchronized getSurface method in the Water class. Before the point is used for comparison a check is performed to ensure that it is not a edge piece of the terrain. If the point is not a boundary value it is compared to its eight surrounding neighbors, to find the lowest point amongst the surface and its neighboring surfaces. The lowest point is determined with the use of nested for loops, and once the lowest point is found a call to the moveWater method in FlowPanel is called to update the image.

Concurrency

Concurrency Design

Thread safety is of utmost importance when designing a multithreaded parallel program. Precautions had to be taken when utilizing the following classes

Class	Reason
Swing	• One of the main reasons for Java Swing is not thread-safe is to simplify the task of extending its components. • Another reason for the Java Swing is not thread-safe due to the overhead involved in obtaining and releasing locks and restoring the state.
Water	The data is shared amongst multiple threads, so maximal concurrency is required to ensure no bad interleaving's or race condition
Terrain	The resource is shared among multiple threads

The Terrain class is only read only, it is immutable that means that the data remains unchanged for the entirety of the simulations run. Since the terrain class is immutable there was no reason to make it thread safe. The swing library carried out tasks mutually exclusive tasks and would not be interfered with, so there was no need for additional protection.

The water class, required protection as multiple threads were reading, writing and modifying shared resources. All data accessors and modifier methods had the synchronized keyword added to its method definition.

The compute directly method synchronized on the water object this prevented access from other threads.

Thread Synchronization

fork() is used to spawn new threads and divide the work. Fork runs subTask1 compute function in a separate thread, whereas subTask2.compute runs the calculation in the current thread. Join( ) is used for synchronization by ensuring that threads wait for work to be completed.

```
int middle= (lo+hi)/2;
Parallel subTask1 = new Parallel(land,height,water,lo,middle);
Parallel subTask2 = new Parallel(land,height,water,middle,hi);
subTask1.fork();
subTask2.compute();
subTask1.join();
```

Liveness and Deadlock

With regards to liveness, the work was divided amongst many ForkJoin threads and the water synchronized methods were made to be very small and not intensive so as to not perform computationally expensive calculations in areas of code that contained locking. As such, the program has a high probability of achieving liveness. With regards to deadlock, the only methods themselves in the Water class could always run to completion and there was no use of deprecated methods such as stop( ) and suspend( ).

Ensuring correctness

I implemented a Sequential and Parallel version of the project and tested both exhaustively. I ran both of my programs side by side with water blocks at the same coordinates on the terrain. On each run the Parallel program matched the outcome of the Sequential program. The simulations were then tested with various combinations of pausing and resuming the runs in order to cause faults, but they still exhibited expected behavior. It was observed that water conservation was preserved i.e. water flowed of the edges and collected in known basins, and no race conditions seemed apparent from the simulation tests. The program was run till 10000 simulation step and no race conditions occurred.

Creativity

- The colour of the water streams was determined by their respective depth value i.e. a lighter shade of blue was assigned to spots that contained less water and a darker shade of blue to spots that contained more water. This adds to the realism of the simulation as it helps visualize the various depths of water flowing across the terrain.
- The java Forkjoin framework was utilized in order to improve performance of the simulation.

Results and Discusions

The program was tested several times using the medium and large sample inputs. The amount of water added to the terrain image was vastly varied and there was no degradation in performance. As hypothesized, it was found that the water displayed expected behavior, that being, collecting in known basins and flowing from high points of elevation(white spots on the grayscale image) to low points of elevation(black/ dark gray spots on the image). The simulation was paused and resumed several times in an attempt to cause a fault, but it was found that it continued to run smoothly. The Sequential cutoff was varied to see the effect on simulation runtime. Improvements were seen for high sequential cutoffs on larger datasets.

Conclusion

The project did indeed require concurrency as the there was access to shared resources. The caveat was deciding the appropriate places to make use of concurrency features. If this wasn't done correctly then the Simulation would not have run as intended and would not display liveliness.