CSC2002S Assignment 2
Concurrent Programming: River Flow Simulation

MBTLIN007 – Lindelani Mbatha

1 Introduction

1.1 Theory

When dealing with large sized problems, sequential processing tends to prove less efficient in some cases. With the development of computer hardware, it is possible to split tasks between multiple threads. However, this raises concerns of data corruption that might arise from multiple threads working on the same data at the same time. As some of the issues that might arise are,

- Race conditions – where the output is unexpectedly dependant on the sequence of some other events within the process.
- Bad Interleaving – where multiple threads operate on the same data but have operations that overlap resulting to interference.

These issues and more other issues introduced by parallel programming or shared data can be solved through Concurrency principles. Some of the notable concurrency principles are,

- Synchronisation – controlling access of multiple threads to a shared resource. This can be done by ensuring that only one thread can operate on the resource at a time.
- Atomicity – To ensure that operations on one piece of data all occur sequentially until it is finished, only making the data available after then.

These principles try to ensure data integrity while trying not to reduce performance. However, when not used properly it can lead to more issues such as deadlocks – where all threads are locked from execution – and reduce liveness of the program.

1.2 Aim

The aim of this assignment is was to implement a multithreaded water flow simulator that shows how water on a terrain flows downhill, accumulating in basins and flowing off the edge at the end of the terrain. The simulation should be demonstrated to the user through an interactive GUI that allows the user to add water on the simulation and allow controls such as play, pause, resent, and end.

This simulation was intended to demonstrate concurrency principles. These are evaluated on their ability to solve data corruption issues and improving performance of the simulation. Furthermore, the implementation should also address issues that might arise with the use of concurrency.

2 Design and Implementation

2.1 Architecture

The general architecture used was the Model-View-Controller (MVC) architecture, which separates the display of information to the internal representation and operations.

The Model is used for data-related logic, and therefor consist of the classes that store data and provide interfaces for manipulating data, the Model classes in this case were,

- Terrain class – which stores all heights at different points on the terrain in an array. This class provides methods to access and manipulates heights. Additional methods were added to tell if a certain point is inside the terrain or not.
- Water class – this store integer units representing depths at different points in the terrain. The depths are stored in an array of the same dimension as that of the Terrain class. Method to access and manipulate these depths can be found on the class. The important methods are inc() and dec() methods which are used to let water flow by incrementing or decrementing the depth by 1.

The View is used for UI logic, and therefor consist of the class responsible for the GUI in this case. The Flow class consist of all UI logic, which consists of JPanels representing the terrain and water, buttons and their action listeners and the mouse adapter and its action listener.

The Controller is an interface between the Model and View components by processing incoming input, manipulate the model and display output using the View. In this case the Controller classes are,

- FlowPanel class – which is responsible for processing inputs from the action listeners from the Flow class. This class is responsible for the continuous painting and repainting of the GUI interface. The class continuously runs the simulation, updating water depths on the terrain and repainting the water until a certain user input is encountered.
- FlowThread – this is a helper to the FlowPanel class. It splits the work of the flow of water on the terrain into 4 threads that each deal with a portion of the water array and flow water by comparing the water surface (water depth + terrain height) at a point to its neighbours and shifting unit depth to the neighbour with the minimum surface.

2.2 Concurrency Consideration

2.2.1 Thread Safety

The classes that consist of shared data across the application had to be thread safe, that is there should be no data corruption issues arising from threads operating on these classes at the same time. The class most prone to data corruption was the Water class, as data in this class kept on being modified by the threads in the FlowThread class. To ensure thread safety, all method that deal with direct access and manipulation of height values were synchronised using the “synchronized” key word in java. This ensures that only a single thread can use this method at a time, making sure that threads will not try to update heights at the same time, and also threads trying to read the heights will always get up to date versions of the height.

2.2.2 Thread Synchronization

Making the Water class thread safe does not solve the problem of asynchronous arrangement of threads and other operations in the program (especially water rendering) which may result to incorrect rendering of water. To solve this, the threads were synchronised using the join() method, which ensures that the program can only continue to the next step of execution once the current thread finishes executing.

A problem of such was encountered before the use of join and the water would only be painted for a finite number of times and then just pause, as the water rendering method kept on accessing the old data.

2.2.3 Liveness and Deadlocks

The threads are always running up until they return as in this case the synchronous keyword was used in such a way that no thread would have to wait for itself resulting to a deadlock. This means that the application and all threads are always live.

2.3 Correctness Evaluation

A quick check was implemented to validate that the system is working and there were minimal and no concurrency issues such as race conditions. The check is based on the principle of water conservation – we cannot create nor destroy water – and so the amount of water added by the user should be kept constant up until the end of the simulation. However, it should be noted that some of the water does flow off the terrain edges.

A variable called *totalWater* is stored in the Water class, this variable gets incremented every time a unit of water is added to the simulation. Another variable called *removedWater* is stored in the Water class and this variable gets incremented every time a unit of water flows off the terrain. Finally, when the user clicks exit a message is printed to the console output the message is either "Water conserved" or "Water not conserved" based on the following comparison,

totalWater – (removedWater + remainingWater) == 0,

where *remainingWater* is the sum of the water in the array at the end of the simulation.

3 Results Discussion

3.1 Medium Sample Results

For the medium sized sample, the program worked as expected. Water was conserved and the performance proved to be good, based on the speed of the water flow, quick response to buttons and no glitches in the water flow.

3.2 Large Sample Results

For the large sample, the program did not work as expected. However, data integrity was maintained as water was conserved. The issue was on performance – there were glitches in the simulation, indicating that there was bad interleaving between the threads updating the water depths and the method dealing with rendering of water.

Even after the ForkJoin Pool platform was used to try and speed up rendering of the water, there was no observable difference from just doing this task sequentially. Furthermore, the repaint() method was called much and much quicker than the depth update rate, but there was still no observable difference.

4 Conclusions

4.1 Bad Interleaving becomes a problem as data size increases

As detailed above, the larger sample's performance was poor due to bad interleaving between updating depths and rendering water. This was not an issue for the medium size data set as there was good performance here. This issue can be solved by speeding up the depths updating process.

4.2 More reduction is needed as the sample size increases

The task had to be conducted with 4 threads only. However, this proved to be less scalable as data size increased. Which means building up on the reduce algorithm used to update depths, reductions must consider data size. A constant split of 4 threads does not work with all data sizes.