CSC2002S
Assignment 4: Concurrency Programming – Tree Simulation
DPLJER001
September 2018

This report details the solution and code for Assignment 4, *Concurrency Programming: Tree Simulation*. Below is an introduction and problem overview, followed by an explanation of the solution architecture and code, the concurrency features used and the architectures' adherence to the MVC design pattern. Results from tests are detailed in Appendix A; these include tests for Race conditions as well as performance tests.

Introduction & Problem Overview

In summary the assignment problem was to simulate large scale forest growth in Java using parallelisation and employing effective strategies to prevent data races, bad interleaving and deadlock when having multiple threads reading and writing from a shared data source. The simulation is graphically redered in a GUI built using the Swing library, hence an additional aim of the assignment was to ensure thread safety where the GUI is concerned by ensuring the simulation code was not being executed on the event dispatch thread (EDT).

The solution to the assignment detailed below is built based on the model view controller (MVC) design pattern, where the shared data sources, the tree and sunlight data as well as a boolean flag (an indicator of whether the simulation is running or not) and year counter, represent the model, the GUI represents the view and the controls are the threads generated by the simulation engine which manipulate the model causing a re-rendering of the view each year/iteration of the simulation.

Description of new Classes and Modifications

The *treeGrow* package was provided as framework code for the solution. Several modifications have been made to the classes provided, some of which are detailed below in logical order. **Fig. 1** illustrates the main components of the solution architecture which is implemented in the code that follows.

In summary, the solution architecture comprises of two main parts:

1. The driver class, treeSimulator, which runs in the main thread. It sets up and controls the GUI, and is also responsible for responding to Action Events which occur when the user presses buttons on the GUI (hence, the event dispatch thread). Since this thread needs to be free to respond to user interaction, the simulation must be run in a separate thread, which is the second part;

2. The simulation engine class simThread, which extends java.lang.Thread, is used to run the simulation in a thread separately to the main class

ensuring thread safety for the Swing library. The simulation runs as a while loop in this class, where each iteration represents a year in which each tree grows by an amount proportional to the sunlight it absorbs in the area it covers on the landscape. The summation of sunlight exposure for all trees is done in parallel using a fork-join pool.

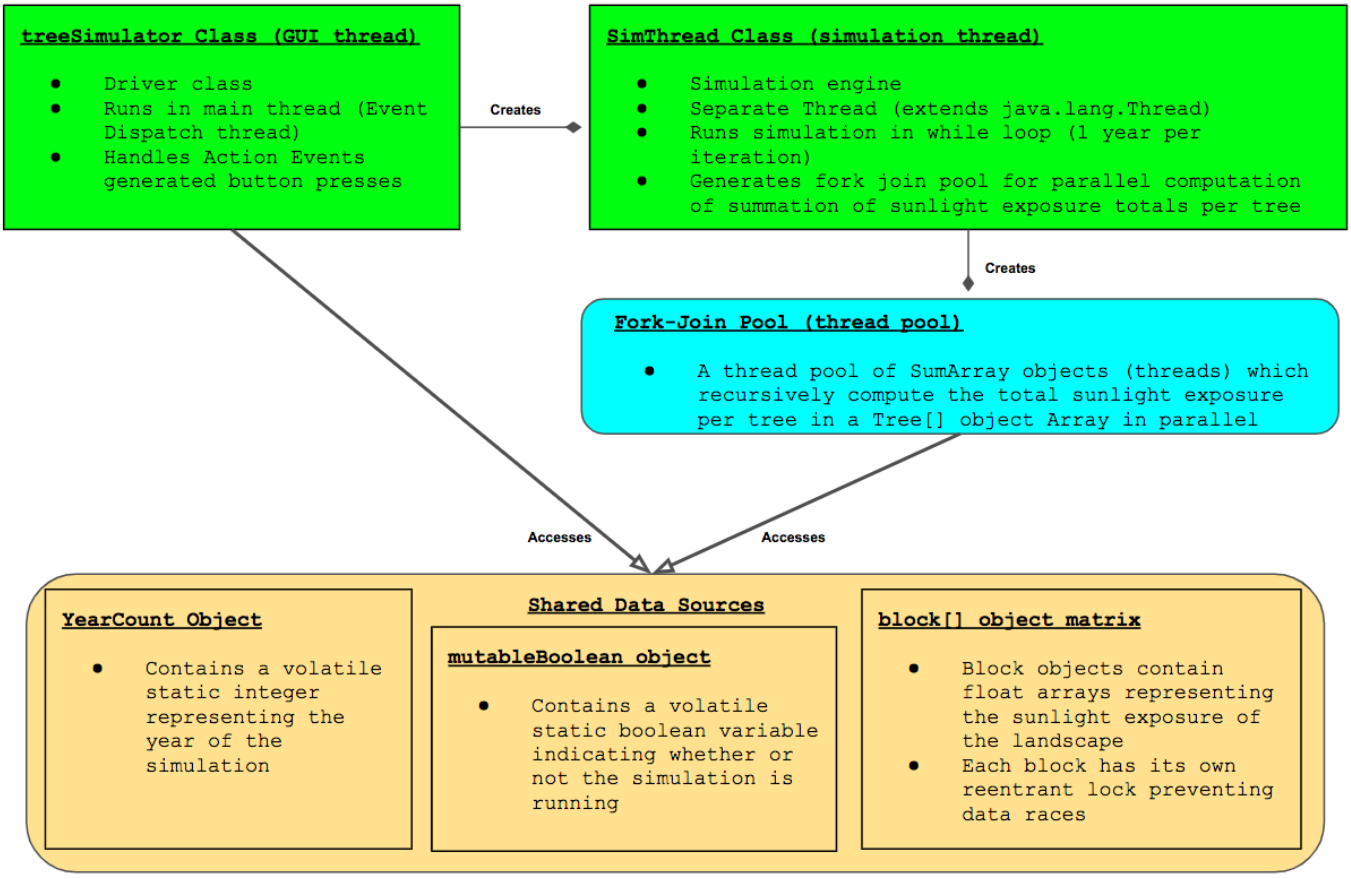


Fig 1. Solution Architecture

The main method of the driver class, *treeSimulator,* is invoked when the program is started. As mentioned above, this class is in charge of setting up and running the GUI and responding to button presses by the user. **Fig. 2** shows the methods associated with each of the buttons and **Fig 3.** shows snapshots of the GUI, taken during simulation. When the user presses the 'play' button, the *PlayPressed()* method is invoked which sets the value of the boolean in the *mutableBoolean* object (*runSim)* to 'true' and creates a new instance of the simulation engine class *SimThread,* calling its *start()* method and creating a new thread for the simulation to run in. The constructor for the *SimThread* object takes a *SunData* object (which came as part of the framework solution code) which contains the array of *Tree* objects as well as the matrix of *block* objects which contain the float matrices representing sun exposure on the landscape (more on this below). It also takes the *YearCount* object containing a static volatile integer variable representing the current year of the

```
public static void ResetPressed(ActionEvent e){
    System.out.println("RESET (in Event Dispatch Thread?: "+javax.swing.SwingUtilities.isEventDispatchThread()+")");
    if(runSim.getValue()){
        jOptionPane.showMessageDialog(frame,  message: "Please pause the simulation before trying to reset.");
    }
    else{
        yearCount.resetCount();
        yearDisplay.setText("year "+yearCount.getCount());
        sundata.resetTrees();
    }
}

public static void PlayPressed(ActionEvent e){
    System.out.println("PLAY (in Event Dispatch Thread?: "+javax.swing.SwingUtilities.isEventDispatchThread()+")");
    runSim.setValue(true);
    sim = new simThread(sundata, yearDisplay, yearCount, runSim);
    sim.start();
}

public static void PausePressed(ActionEvent e){
    System.out.println("PUASE (in Event Dispatch Thread?: "+javax.swing.SwingUtilities.isEventDispatchThread()+")");
    runSim.setValue(false);
}

public static void EndPressed(ActionEvent e){
    System.out.println("END (in Event Dispatch Thread?: "+javax.swing.SwingUtilities.isEventDispatchThread()+")");
    System.exit( status: 0);
}
```

Fig 2. Action Event Methods For Buttons

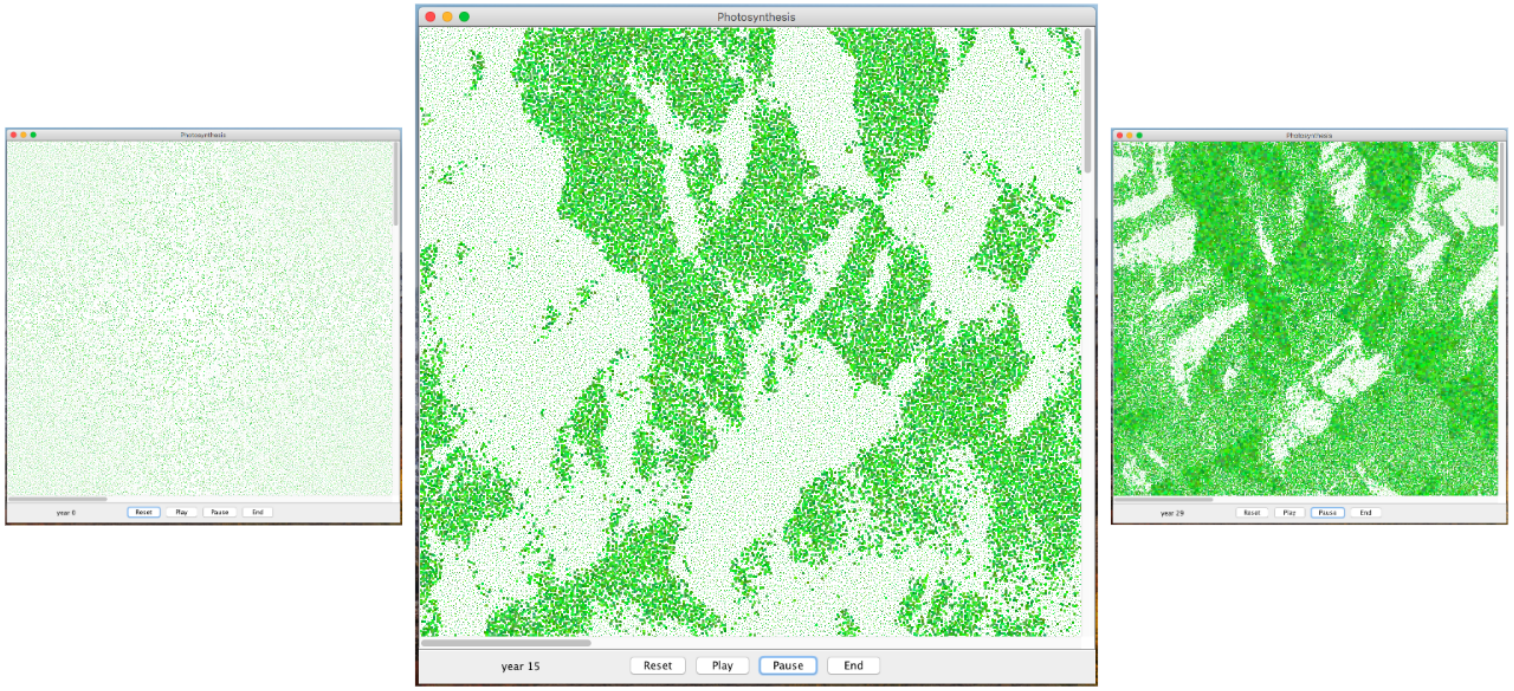


Fig 3. GUI snapshots during simulation

simulation, the JLabel object representing the year count on the GUI and a mutableBoolean object indicating whether the simulation is currently running or not. When invoked, the *run()* method inside the *simThread* class, seen in **Fig**

4., runs a while loop which checks the mutableBoolean object is set to 'true' before each cycle (year) of the simulation. Each year of the simulation, the Tree[] objects are sorted into extent groups (bands) and each selection is passed into a *forkjoinpool*, *SumArray* (which extends *RecursiveAction)*, which recursively creates new threads of work to sum up the sunlight exposure per tree. The first band of trees processed are trees with extents greater than 20. By design, no tree can grow to an extent larger than 20 to prevent infinite growth, but in order to accurately simulate the real world scenario of forest growth, the processing of trees with an extent of 20 or greater means that the larger trees still absorb the sunlight in the area they occupy, just without growing. In this way smaller trees 'beneath' larger ones still get less sunlight. At the end of each simulation cycle, the method *sundata.sunmap.resetShade()* is called to reset the sunlight values in the float matrices contained in the block objects for the next year and the *yearCount.plusCount()* increments the year count by one which updates on the GUI. Below is an examination of the *SumArray* and *block* classes which details how sunlight values are read and written from the shared data source.

```
    public void run(){
        ArrayList<Tree> treeSelection;
        Land sunMap = sundata.sunmap;
        Tree[] trees = sundata.trees;
        double avgTime = 0.0;
        System.out.println("Running Simulation (in Event Dispatch Thread?: "+javax.swing.SwingUtilities.isEventDispatchThread()+")");


        while(runSim.getValue()){ // while the simulation is running: one loop per year

            float maxEx = 100.0f; // very tall trees still absorb shade even though they do not increase their extent beyond 20
            float minEx = 20.0f;

            treeSelection = getTreeSelection(minEx, maxEx, trees);
            sum(treeSelection, sunMap);

            maxEx = minEx;
            minEx = maxEx-2;

            for(int i=0; i<10; i++){
                treeSelection = getTreeSelection(minEx, maxEx, trees);
                sum(treeSelection, sunMap);
                maxEx = minEx;
                minEx = minEx - 2;
            }

            sundata.sunmap.resetShade();
            year.setText("year "+ yearCount.getCount());
            yearCount.plusCount();
        }
    }

    private ArrayList<Tree> getTreeSelection(float min, float max, Tree[] trees){
        ArrayList<Tree> toReturn = new ArrayList<>();
        for(Tree t : trees) {
            if(t.inrange(min, max)){
                toReturn.add(t);
            }
        }
        return toReturn;
    }

    private void sum(ArrayList<Tree> treeList, Land sunMap){
        fjPool.commonPool().invoke(new SumArray( l: 0, treeList.size(), treeList, sunMap));
    }
}
```

Fig 4. The run(), getTreeSelection() and sum() method in the *simThread* class

```java
public class SumArray extends RecursiveAction { // returns floating point value (a sum)

    private int lo; // point to start summing in array
    private int hi; // point to end summing in array
    private ArrayList<Tree> treeList;
    private Land sunMap;

    SumArray(int l, int h, ArrayList<Tree> tl, Land s){
        lo = l;
        hi = h;
        treeList = tl;
        this.sunMap = s;
    }

    @Override
    protected void compute(){

        if ((hi - lo) < constants.SEQUENTIAL_CUTOFF) {  // minimum tasks a thread must handle
            sequentialSum();
        }

        else{
            SumArray left = new SumArray(lo, h: (hi+lo)/2, treeList, sunMap);
            SumArray right= new SumArray( l: (hi+lo)/2, hi, treeList, sunMap);

            left.fork(); // branch off thread
            right.compute(); // main thread does this
            left.join(); // wait until left is done
        }
    }
```

Fig 5. The *sumArray* class (part A)

```java
private void sequentialSum(){
   for(int i=lo; i<hi; i++) {
       float avg = 0.0f;
       Tree tempTree = treeList.get(i);

       int xCenter = tempTree.getX();
       int yCenter = tempTree.getY();
       int treeExt = Math.round(tempTree.getExt());

       // get the boundaries of a tree on the landscape
       int topRow = Math.max(0, yCenter-treeExt);
       int bottomRow = Math.min(sunMap.getDimY() - 1, yCenter+treeExt);
       int leftCol = Math.max(0, xCenter-treeExt);
       int rightCol = Math.min(sunMap.getDimX()-1, xCenter+treeExt);

       // find out which blocks are occupied by the corners of each tree (and hence all blocks with which the tree intersects)
       block topLeftBlock = sunMap.getBlock(topRow, leftCol);
       block topRightBlock = sunMap.getBlock(topRow, rightCol);
       block bottomLeftBlock = sunMap.getBlock(bottomRow, leftCol);
       block bottomRightBlock = sunMap.getBlock(bottomRow, rightCol);

       // lock blocks in order of their unique block number (impose global order and prevent deadlock)
       block[] occupied_Blocks = {topLeftBlock, topRightBlock, bottomLeftBlock, bottomRightBlock};
       Arrays.sort(occupied_Blocks);

       //lock blocks in order
        for(int n=0; n<4; n++){
            occupied_Blocks[n].lock.lock();
        }

        float count = 0.0f;
       for(int row=topRow; row<=bottomRow; row++){
           for(int col=leftCol; col<=rightCol; col++){
               float val = sunMap.getBlockValue(row, col);
               avg += val;
               count++;
           }
       }

       //unlock blocks in order
       for(int n=0; n<4; n++){
           occupied_Blocks[n].lock.unlock();
       }

       // feed sunlight to tree to tree and diminish sunlight value in occupied cells
       avg = avg/count;
       tempTree.sungrow(avg);
   }
}
```

Fig 6. The *sumArray* class (part B)

Fig. 5 and **Fig. 6** show the *sumArray* class which essentially creates threads of work by recursively halving the array of Tree[] objects in a band until the SEQUENTIAL_CUTOFF constant of 500 trees is reached, at which point the *sequentialSum()* method is called.

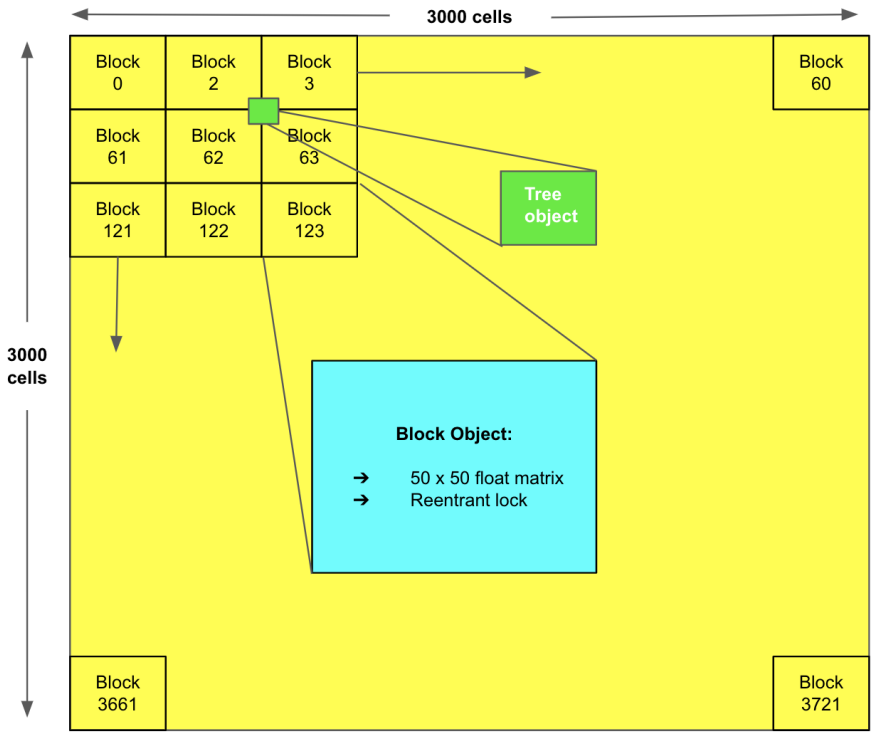


Fig 7. Square landscape of sunlight values split into 'blocks'

As illustrated in **Fig. 7**, in the simulation, a 'tree' is square and can have a width of at most 41 cells (an extent of 20, plus the center cell), and the block objects containing the square float matrices have been set to have dimensions of 50x50. Hence, a tree may extend into at most 4 blocks, with each of the four corners of the tree being the farthest reach of the tree from it's center. Inside the *sequentialSum()* method the boundaries of the tree are calculated and then the blocks occupied by the tree are calculated by the *sunMap.getBlock()* method which is in the *Land* class. The block address of a particular cell is calculated by performing integer division of the (x,y) coordinate pair by the dimension of the block (50), and then the cells address inside the block is calculated by taking the modulus of the (x,y) coordinate pair with the dimension of the block (50). For example, a cell located at (62,85) is located at block (1,1) and inside the block the cell has the coordinates (12, 35). Once the relevant four blocks occupied by the tree have been located (and there may be repetitions of the same block), they are assigned to variables, placed in an array and sorted according to their block numbers (see **Fig. 8**). The blocks are then locked in order - this global ordering safeguards the program against deadlock, where two threads could

potentially get stuck in a mutual deadly embrace (each thread waiting to acquire a lock that the other has) if locks were acquired randomly. Once all four block objects have been locked, the average sunlight for the tree is calculated, and the tree extent is increased accordingly (provided it's extent is less than 20). We will now take a closer look at the block object:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class block implements Comparable<block>{
    volatile float [][] values;
    private int blockNum;
    Lock lock = new ReentrantLock();    // lock used in the sequentialSum() method

    public block(int size, int num){
        values = new float[size][size]; // 50 by 50 float array of sunlight values
        blockNum = num;                 // unique block number
    }

    public void setValue(int y, int x, float val){
        values[y][x] = val;
    }

    public float getValue(int row, int col){
        float temp = values[row][col];
        values[row][col] = temp * 0.1f; // reduce sunlight cell to 10% of their original value
        return temp;
    }

    public int getBlockNum() {
        return blockNum;
    }

    // Allows blocks to be sorted by their block numbers when locking
    public int compareTo(block other){
        return Integer.compare(this.getBlockNum(), other.getBlockNum());
    }
}
```

Fig 8. The *block* object

As stated above, a block object contains a 50x50 sub section of the 3000x3000 sunlight matrix representing the sun exposure of the simulated landscape. Access to this shared data by threads can only be granted when a the ReentrantLock of the block has been acquired, and locks for multiple blocks are always acquired by order of block numbers. The sorting of blocks by their unique block numbers is possible since the block class implements the Comparable interface, allowing an array of blocks to be sorted using *Arrays.sort(blockArray)*.

After each tree has had it's total sun exposure summed for a single year, it's extent is increased and if the increase takes the tree from an extent of x.4 to x.5 (where x is an integer, and 0 <= x < 20) then it's extent is increased visually in the GUI from x to (x+1), i.e. the integer gets rounded off. After

a year of simulation the sunlight values in the block object matrices get reset, and the simulation continues onto the next year. This is the essence of the simulation solution code. Classes which have not been displayed and discussed above for the sake of reasonable report length are; the *SunData*, *Land*, *Tree*, *YearCount*, *mutableBoolean* and *forestPanel* classes. All of these are in the /**src** folder, and are relatively self-explanatory given the information above.

Concurrency Features Used

The points below highlight some key concurrency features of the solution architecture discussed above.

Thread safety for the Swing Library:

- As stated above, the simulation engine is purposefully run in a separate thread (the simThread class) to the main thread (event dispatch thread) in order to ensure the GUI remained responsive to button pushes by the user, and rendering was as smooth as possible.

- The year count variable used to track the year of the simulation and update the GUI label was made *static volatile* to ensure it was read from and written to main memory so it updated accurately in the GUI each iteration.

- Similarly, the boolean flag variable in the mutableBoolean class was also made to be *static volatile*.

Thread safety for shared data between threads:

- The ordered locking of the reentrant locks of block objects shared by threads (containing the float matrices which encode sunlight values) ensures:
    - Safety against race conditions
    - Safety against bad interleaving
    - Safety against deadlock, since the blocks are locked according to a global ordering.

Thread synchronization:

- The method of synchronizing threads was not used since the use of Reentrant locks would render it redundant.

Liveness of simulation:

- The parallel computation of sunlight totals per tree using the fork join pool ensured the computation is sufficiently fast for the visual simulation to perform with fluidity.

- In fact, the simulation calculations occur at a rate faster than the Swing library is able to render the simulation visually, so in the code submitted a Thread.sleep() method has been called in each iteration of the simulation to allow the GUI time to render properly with each update (without this the program can simulate 30 years in a few seconds).

Validate system/check for errors:

Appendix A shows results of a test simulation run using only two trees, each run in their own thread, and a landscape area of 60x60 to test for race conditions. Results from the test (as illustrated in the Appendix) show that when the two trees reach extents where they occlude each other they read and write to the shared memory as expected, due to the order locking algorithm, and no race conditions or bad interleaving occur.

Performance of the simulation relative to block size (granularity):

It is important to ensure that the size of the blocks used in the algorithm above ensures that the program is running as efficiently as possible. Appendix A shows the raw data from the performance tests done to measure this; an average time per year of simulation was measured over a 50 year simulation run for block sizes of {50, 200, 400, 800, 1500, 3000}. **Fig. 9** illustrates the results of the test:

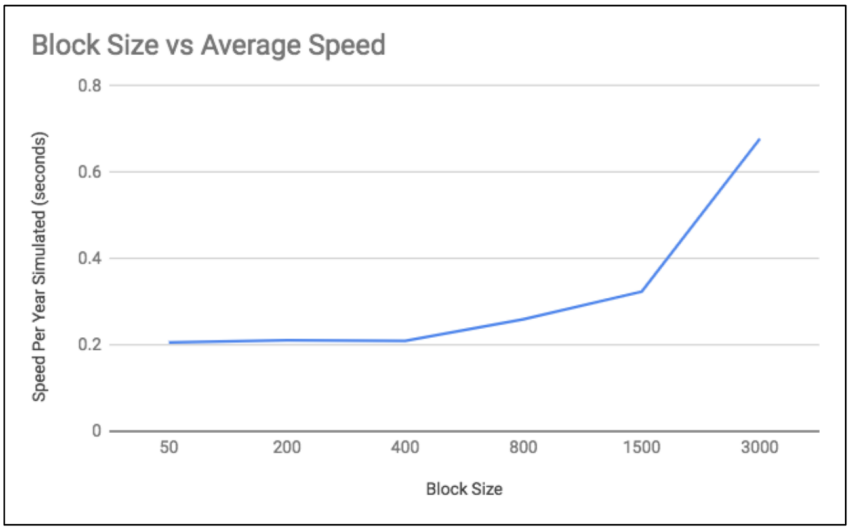


Fig 9. Average time per year simulated Vs. block size

As expected, as the granularity of the block size approaches the size of the landscape (3000x3000) the time per simulated year increases steeply. A smaller granularity ensures maximum performance benefit is obtained from the parallel algorithm.

Model-View-Controller Components

It is worth clarifying how the solution architecture conforms to the MVC design pattern. Below is an outline of the how the three components manifest in the solution code:

- Model:
    - The Tree[] object array represented as a forest of trees in the view (GUI), manipulated by the controller (the threads which ultimately update the tree extents)
    - The Boolean flag (*mutableBoolean* class) indicating whether or not the simulation is running or not.
    - The year counter (*YearCount* class) representing the year of the simulation, displayed as a label in the GUI (view)

- View:
    - GUI; the Tree[] objects and year counter visually illustrating the forest growth simulation as green squares which grow over time.
    - Updates/re-rendering pauses when the boolean flag variable is set to false.

- Controller:
    - The threads generated by the simulation engine which manipulate the model; increasing the extent of each Tree[] object each iteration of the simulation, casing the view to update.
    - The simulation engine runs directly as a result of a user pushing the 'Play' button ('Reset' and 'Pause' have direct effects on the state of the view and model, too), hence the buttons may also be thought of as part of the 'controller'.

Concluding Remarks

The solution program has been designed to run as efficiently as possible by making maximal use of parallelisation in Java, while also using appropriate concurrency safety methods to ensure Thread safety for the shared variables as well as the Swing library. Appropriate testing was conducted to ensure the code was not at risk of permitting race conditions, bad interleaving or deadlock, as well as to ensure that the granularity of the blocks (block sizes) were set to produce optimal parallel performance. Finally, the program architecture adheres to the MVC design pattern.

Appendix A: Test Results

Test: Data Races

The data used for the test simulation comprised of two trees in a 60x60 grid positioned 1 cell apart. The SEQUENTIAL_CUTOFF variable is set to less than 2 to ensure the tree calculations run in separate threads. The details reported by the program are:

Dimensions of land (x,y): (60, 60)
Block size (square): 50
Dimension of block matrix (x,y): (2, 2)
Number of trees: 2

For the first fourteen years of the simulation the two trees do no occlude each other as their extents remain below 0.5, as illustrated by the image below, alongside the output from the test program, reporting a summary of each tree per year as well as the values of the sunlight cells absorbed by each tree:

Year: 0

Tree 0 pos: 20 20 sun: 8.0 Ext: 0.408
Tree 1 pos: 20 22 sun: 7.0 Ext: 0.407

Tree: 0
8.0

Tree: 1
7.0

Year: 1

Tree 0 pos: 20 20 sun: 8.0 Ext: 0.41599998
Tree 1 pos: 20 22 sun: 7.0 Ext: 0.414

Tree: 0
8.0

Tree: 1
7.0

.
.
.

Sample from sun matrix

	19	20	21	22	23
19	5.0	7.5	4.0	...	...
20	5.5	Tree 0 (8.0)	4.0	...	...
21	3.5	6.5	4.0	...	...
22	4.5	Tree 1 (7.0)	4.5	...	...
23	4.0	7.5	5.0	...	...

At year 15, both trees now gain an extent of greater than 0.5, and hence occlude each other in 3 of the 9 cells which each tree covers: (19, 21), (20, 21) and (21, 21) as seen in the diagram below. As is evident from the data

below, since both trees are in the same extent band (0.0 to 2.0) each tree "competes" for the right to occlude the other and get the majority (90%) of the sunlight in the shared cells. In Year 15 and 16 tree 1 claims the majority of the sunlight, and the cells are reduced appropriately and absorbed by tree 0 afterwards. In year 21 (shown below) the opposite occurs and tree 0 claims the majority of the sunlight as it's thread got to the processor ahead of the thread belonging to tree 0. This shows the program works are expected by locking the block and giving one tree access to all the cells exclusively before unlocking it, avoiding any possibility of data races.

Year: 15

Tree 0 pos: 20 20 sun: 35.4 Ext: 0.6353998
Tree 1 pos: 20 22 sun: 46.5 Ext: 0.5515

Tree: 0
5.0 7.5 4.0
5.5 8.0 4.0
0.35 0.65 0.4

Tree: 1
3.5 6.5 4.0
4.5 7.0 4.5
4.5 7.0 5.0

Sample from sun matrix

	19	20	21	22	23
19	5.0	7.5	4.0	...	...
20	5.5	Tree 0 (8.0)	4.0	...	...
Overlap → 21	3.5	6.5	4.0	...	...
22	4.5	Tree 1 (7.0)	4.5	...	...
23	4.0	7.5	5.0	...	...

Year: 16

Tree 0 pos: 20 20 sun: 35.4 Ext: 0.67079985
Tree 1 pos: 20 22 sun: 46.5 Ext: 0.59800005

Tree: 0
5.0 7.5 4.0
5.5 8.0 4.0
0.35 0.65 0.4

Tree: 1
3.5 6.5 4.0
4.5 7.0 4.5
4.5 7.0 5.0
.
.
.
Year: 21

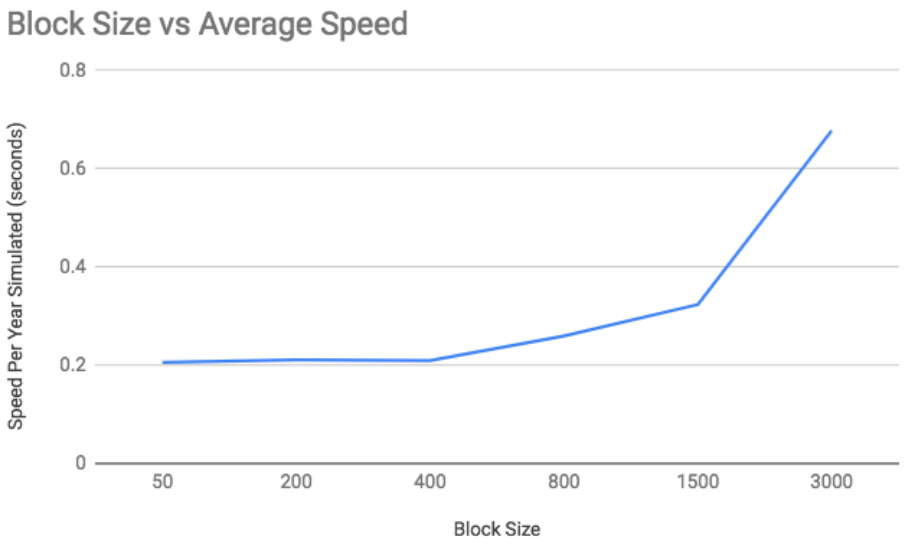
Tree 0 pos: 20 20 sun: 48.0 Ext: 0.8729999

Tree 1 pos: 20 22 sun: 33.9 Ext: 0.8053002

Tree: 0
5.0 7.5 4.0
5.5 8.0 4.0
3.5 6.5 4.0

Tree: 1
0.35 0.65 0.4
4.5 7.0 4.5
4.5 7.0 5.0
.
.
.

Test: Performance Vs Block Size



Data output from program:

```
Loading Data from file data/sample_input.txt
Dimensions of land (x,y): (3000, 3000)
Block size (square): 50
Dimension of block matrix (x,y): (61, 61)
Number of trees: 1000000
Data loaded
PLAY (in Event Dispatch Thread?: true)
Running Simulation (in Event Dispatch Thread?: false)
```

Average time taken for 50 year simulation: **0.20597 seconds**

Loading Data from file data/sample_input.txt
Dimensions of land (x,y): (3000, 3000)
Block size (square): **200**
Dimension of block matrix (x,y): (16, 16)
Number of trees: 1000000
Data loaded
PLAY (in Event Dispatch Thread?: true)
Running Simulation (in Event Dispatch Thread?: false)
Average time taken for 50 year simulation: **0.21086000099778177**

Loading Data from file data/sample_input.txt
Dimensions of land (x,y): (3000, 3000)
Block size (square): **400**
Dimension of block matrix (x,y): (8, 8)
Number of trees: 1000000
Data loaded
PLAY (in Event Dispatch Thread?: true)
Running Simulation (in Event Dispatch Thread?: false)
Average time taken for 50 year simulation: **0.20944000005722047**

Loading Data from file data/sample_input.txt
Dimensions of land (x,y): (3000, 3000)
Block size (square): **800**
Dimension of block matrix (x,y): (4, 4)
Number of trees: 1000000
Data loaded
PLAY (in Event Dispatch Thread?: true)
Running Simulation (in Event Dispatch Thread?: false)
Average time taken for 50 year simulation: **0.2594400005042553**

Loading Data from file data/sample_input.txt
Dimensions of land (x,y): (3000, 3000)
Block size (square): **1500**
Dimension of block matrix (x,y): (3, 3)
Number of trees: 1000000
Data loaded
PLAY (in Event Dispatch Thread?: true)
Running Simulation (in Event Dispatch Thread?: false)
Average time taken for 50 year simulation: **0.32365999966859815**

Loading Data from file data/sample_input.txt
Dimensions of land (x,y): (3000, 3000)
Block size (square): **3000**
Dimension of block matrix (x,y): (2, 2)
Number of trees: 1000000
Data loaded

```
PLAY (in Event Dispatch Thread?: true)
Running Simulation (in Event Dispatch Thread?: false)
Average time taken for 50 year simulation: 0.6773599994182586
```