
Introduction

In this program, an array of sun data (sun map) is provided which indicates the average amount of sun received by a particular area of land. A data set of tree canopies and their locations on the sun map are also provided. The program must determine how much each tree grows (both horizontally and vertically) each year using the average sunlight that the tree gets to do so. As the trees grow, they will start to occlude each other and thus a tall tree will cause a shorter tree beneath it to get less sunlight and the program must take this into account. Many of the calculations will be done in parallel and some will try to access the same part of the sun map at the same time and so implementation of concurrency is the main aim of this assignment.

Overview of the workings of the program

The main program, TreeGrow, will start two threads. One to handle the rendering of the GUI and another to handle the calculations of the extents of all the trees called Simulator. The simulator will run in a loop with each iteration representing one year. For each year, the program will iterate sequentially through the different possible layers that trees could be in starting with the highest layer of (18, 20] and working down to (0, 2]. For each layer, only the trees in that layer are processed and thus it is ensured that trees in different layers are not processed simultaneously.

For each layer that is processed, a fork/join algorithm is used by invoking the SunThread class. Within this class, the array of Trees is split across multiple threads (the total number of threads is determined by the sequential cutoff) and within these threads, the calculations are done for each Tree. These calculations involve accessing the global sunmap array so that the average amount of sunlight for the tree can be calculated, the calculated value is used to grow the extent of the Tree and is also used to update the global sunmap array. Since there will be Trees processed in parallel, it could be the case that two or more threads are trying to read and write to the same part of the sunmap array at the same time which could result in corruption of the data. This is a problem for Trees that are in the same layer that overlap each other because what could happen is that some blocks of the one Tree is processed before the same blocks of the second Tree but then also some of the blocks of the second Tree that are overlapping with the first Tree are processed before the same blocks of the first Tree. Thus the result will be as if the two trees are interleaving which is not what we want. It is necessary that one of the Trees processes the overlapping parts completely before the second Tree starts and that the second Tree uses the values of the sunmap after the first Tree's shadows have been taken into account.

In order to ensure that overlapping trees in the same layer are always processed one after the other, I created a class called LandChunks which is essentially just a wrapper class for a 2D array of floats. When the Land class is created and the sunmap array is complete with all the sun values, I create a new 2D array of type LandChunks called splitSunArr and then I split the sunmap array into smaller 2D array of type LandChunk and save each in the splitSunArr array. Then each time a Tree is being processed, I locate the location of the 4 corners of the tree and determine which of the smaller 2D arrays the corners are located and then before the tree is processed, a lock is placed on those 4 (could be fewer depending on where the tree is located) LandChunk objects. During the time that the Tree is being processed, no other Tree in the same blocks will be able to continue until the first Tree is complete. But all other Trees in different locations that do not occupy the same blocks will be able to continue processing.

Description of classes

LandChunk class

The LandChunk class is a class that I added that stores in it a two-dimensional array of floats. It was necessary for me to do this as later on in the code when locks are used to lock on a specific piece of land, it will be objects of type LandChunk that will be locked on. This class contains a constructor and getters and setters that can be used to read or modify the values within the arrays.

Land class

The Land class stores one two-dimensional float array called “sunArrV1” that is used to store the original sun values of the sun-map and another two dimensional array of type LandChunk that will be used to store LandChunk variables. The purpose of this 2D-array is to allow the original sun-map to be broken up into smaller pieces so that when processing is done on the map when Trees are being processed, parts of the array will be blocked for a certain Tree while other Trees at different parts of the array will still be able to process.

Figures 1 and 2 below show an example of the float sunArrV1 and the corresponding LandChunk splitSunArr that would result.

Figure 1 show that the sunArrV1 array is a 9x9 array of floats and Figure 2 shows that the splitSunArray is a 3x3 array of type LandChunk, each of which contains a 3x3 float array

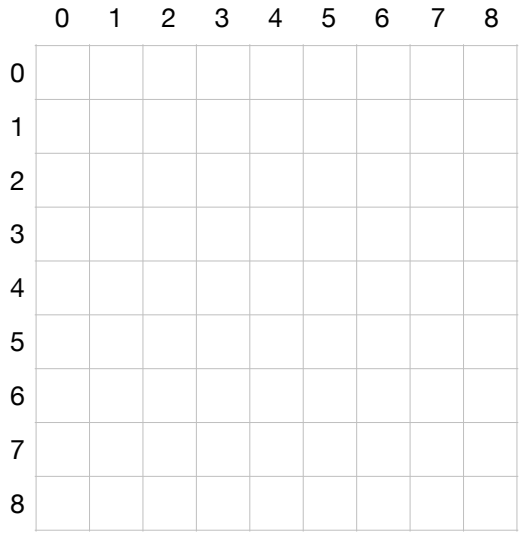

Figure 1

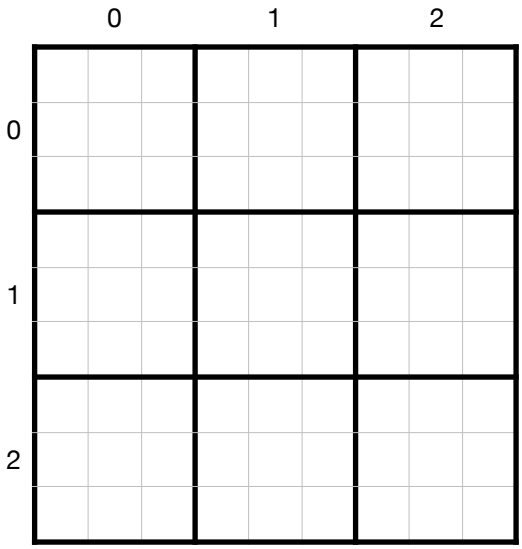

Figure 2

The Land class contains getter and setter methods for the LandChunk splitSunArr array as well as a method to reset the values in the blocks of the splitSunArr to match the original values in the sunArrV1.
The Land class also contains a method called “shadow” which takes a Tree object as a parameter and uses the dimensions of the Tree to update the values in the splitSunArray array. And the Land class also has various get and set methods for returning or setting a certain sun value at a certain position in the sun-map

Tree class

The tree class is used to create a Tree object for each tree in the data set and for each Tree it stores the position of the centre of the Tree as well as the extent of the Tree. The class contains getter and setter methods for the trees extent, as well as a method called “sungrow” that, when called, will use the Tree’s position and extent to determine how much the tree’s extent should grow by. This method will also update the values in the sun-map dataset. I removed the original method called “sunexposure” that would purely be used to calculate the total sun the the particular Tree gets. Instead I moved this computation to within the “sungrow” method. I will explain why I did this in the concurrency section of the report but in essence, it was necessary to atomize the process of calculating the sun exposure of a tree and then updating the sun map accordingly. This is easier to do when all the relevant code is in the same method so that the same lock can be used while doing both of these operations.

Simulator class

The Simulator class extends the java.lang.Thread class and it is called (.start()) from the TreeGrow classes main method. This class controls the computations within the program. It runs in a loop where every iteration of the loop represents a new year and for every year the program loops through all the layers of the Trees and for each layer, it initializes and invokes a SunThread object which will use the fork-join algorithm to calculate the computations for each tree. The Simulator program is also responsible for responding accordingly when any of the buttons on the GUI are pressed.

SunThread class

This class extends the RecursiveAction class. It is responsible for using the fork-join method to break the Tree array up in to smaller array until the array sizes are smaller than the sequential cut-off. Then for each Tree that is in the correct layer being calculated, the “sungrow” method is called on the Tree. This is the method in the Tree class that was mentioned before that is responsible for accessing values in the sun-map dataset to calculate the average sun exposure for a Tree and then to update the sun-map values accordingly.

TreeGrow class

This class contains the main method of the program. In it, the GUI is initialized and the rendering thread as well as the Simulation thread are started. During the GUI setup, I added the buttons and an ActionListener for each button.

Concurrency

There are two main shared variables in this program. The first being the sun-map data array as well as the array of Trees.

The danger with the shared sun-map data array is that two or more threads processing overlapping Trees in the same layer will try to read and write to the sun-map data array at the same time and will thus cause a data-race. This would cause the Tree extents to be updated incorrectly and thus would influence the accuracy of the program. In order to prevent this from happening, I split the single 2D sun-map array into smaller 2D arrays each of which I store in a LandChunk object. Then each time a thread processed a certain Tree, it is first required to block the 4 LandChunk objects that the Trees corners are in. This was done using synchronize blocks. Figure 3 below shows a visual representation of this. Tree T1 has corners 4 of the LandChunk objects (yellow) and so the processing for the Tree will only happen once all 4 of the LandChunk objects have been locked. Tree T2 has all of its corners in 2 LandChunk objects (blue) and so these 2 objects will need to be blocked before the Tree can be processed. Note that the processing for Tree T3 will not be able to continue until the thread processing T1 has released the locks on the LandChunk objects. Thus while Tree T1 is being processed by a

thread, the thread processing Tree T3 will not be able read from or write to the sun-map. This means that for Trees that are in the same layer and are overlapping, one of the Trees will complete its processing (read and writes to the sun-map) completely before the second Tree starts. This also means that Trees in different parts of the array can access the sun-map data structure safety at the same time since they do not occupy the same blocks and so their independent processing is not affected. This means that in the example shown below in Figure 3, Tree T1 and tree T2 would be able to process at the same time.

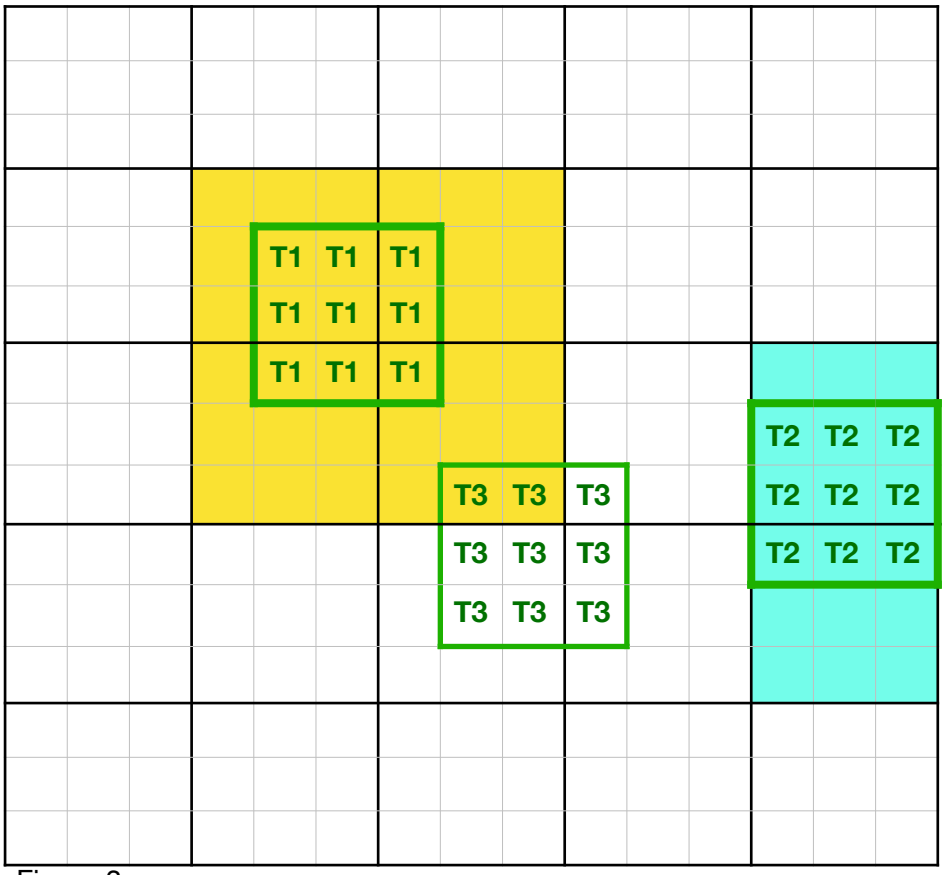

Figure 3

Since different threads cannot access the same points in the sun-map data structure at the same time, the data structure is thread safe.

With this thread safe structure in place, it was not necessary to synchronize any of the Land classes getter or setter methods since it is now guaranteed that there will never be two threads trying to use these methods to access and write to the same part of the sun-map dataset.

When the LandChunk objects are being locked on, it is done in a clock-wise order starting from the top left block. Because of this universal order in which the LandChunk objects are locked and unlocked, there is never a case of deadlock where two Trees that share a LandChink clock will be waiting for the each other.

Another shared variable is the array of Trees. The ForestPanel uses the Tree array to determine how large the squares representing each tree should be drawn. This is done using the Tree classes .getExt() method.

Initially the ForestPanel had the following code (Figure 4) in it that it would use to determine how large each tree block should be drawn:

```
for(int layer = 0; layer <= 10; layer++) {
    for(int t = 0; t < forest.length; t++){
        int rt = rndorder.get(t);

        if(forest[rt].getExt() >= minh && forest[rt].getExt() < maxh) { // only render trees in current band
            // draw trees as rectangles centered on getX, getY with random greenish colour
            g.setColor(new Color(rnd.nextInt(100), 150+rnd.nextInt(100), rnd.nextInt(100)));
            g.fillRect(forest[rt].getY() - (int) forest[rt].getExt(), forest[rt].getX() - (int) forest[rt].getExt(),
                    2*(int) forest[rt].getExt()+1,2*(int) forest[rt].getExt()+1);
        }
    }
    minh = maxh;  // next band of trees
    maxh += 2.0f;
}
```

Figure 4

The problem that arrises here is that the Simulator thread which is running separately to this thread constantly updating the Trees extent variables. So what could happen here is that the if-statement is passed due to the extent variables that it accesses but then before it accesses the extent variables again within the if-statement, the other Simulator thread might have changed the extent variable for the Tree in question such that it should no longer pass the if-statements conditional statement.

In order to fix this, I adjusted the code as seen in Figure 5 below.

```
for(int layer = 0; layer <= 10; layer++) {
    for(int t = 0; t < forest.length; t++){
        int rt = rndorder.get(t);
        float extent = forest[rt].getExt(); //local copy of extent variable
        if(extent >= minh && extent < maxh) { // only render trees in current band
            // draw trees as rectangles centered on getX, getY with random greenish colour
            g.setColor(new Color(rnd.nextInt(100), 150+rnd.nextInt(100), rnd.nextInt(100)));
            g.fillRect(forest[rt].getY() - (int) extent, forest[rt].getX() - (int) extent,
                    2*(int) extent+1,2*(int) extent+1);
        }
    }
    minh = maxh;  // next band of trees
    maxh += 2.0f;
}
```

Figure 5

What I have done in the code above is saved the specific Tree's extent variable locally and then use the local copy (which will not be changes by the Simulator class) to evaluate the if-statements condition and then to do the necessary Tree block drawing.

Validation

To validate that overlapping trees in the same layer are not processed at the same time and that there are no race conditions, I created my own small sample data (illustrated in Figure 4 below) of a 9x9 sun-map containing two overlapping trees (T1: x = 5, y=3, extent = 1, T2: x = 4, y=5, extent = 1). From this sample, I can manually calculate what the expected outcomes should be and then I can check that using the appropriate locks in my program, ensures that the output of the program is as expected.

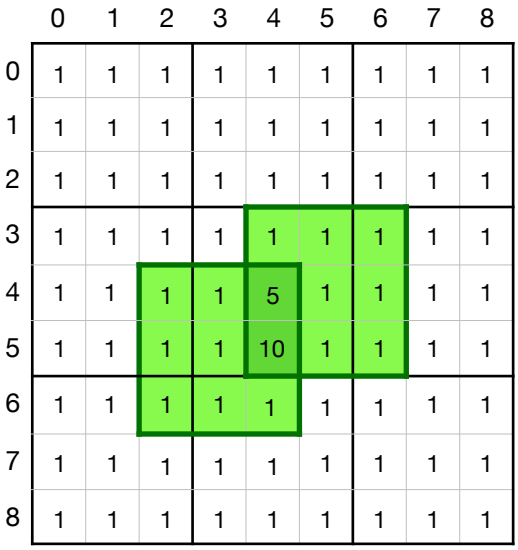

Figure 4

The expected, correct output is as follows: the two trees are overlapping and so one of them needs to be processed and should manipulate the sun-map data before the other Tree starts. Thus is Tree T1 starts first, it should calculate that its average sunlight is s1 = (1+1+5+1+1+10+1+1+1)/9 = 2.444 and then it should reduce the values of the sunmap of the parts it is covering by 10%. This would then result in Tree T2 calculating its average sunlight as s2 = (1+1+0.5+1+1+1+1+1+1)/9 = 0.9444. A similar outcome would result if T2 was processed before T1. An incorrect result would be if both Trees were processes at the same time and both used the initial values of the sun-map to determine there sun-exposure. Both threads could then also try to write to the sun-map data set at the same time and this is a data race.

Figure 5 below shows the output from my program when no locks were in place and Figure 6 shows the output from the program when locks were in place. Both of the outputs show the results of the initial average sunlight calculations for the Trees. It is clear that when no locks are used, a race condition occurs and that the thread processing the second Tree is using incorrect values from the sun-map data set. When the locks are used, it can be see that the calculated averages for the trees are as expected.

```
new year
average sunlight: 2.4444444
average sunlight: 2.4444444
```

Figure 5: output when no locks are used

```
new year
average sunlight: 2.4444444
average sunlight: 0.9444444
```

Figure 6: Output when locks are used

Model-View-Controller

The Model-View-Controller pattern requires that the internal representations of the information (model) used in the program are separate from the display of the information (view). Another separate part is the controller which is used to communicate information regarding user actions to manipulate the model and thus update the view. Figure 7 below shows a graphic interpretations of the concept.

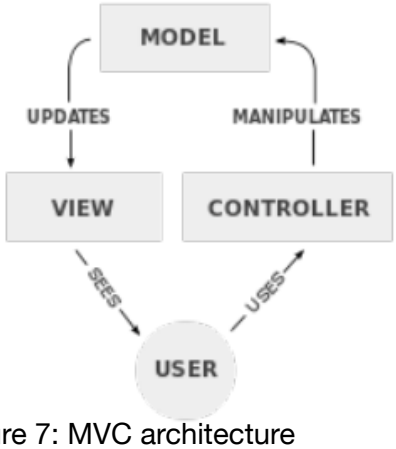

Figure 7: MVC architecture

In my program, the SunData, Land, Tree and LandChunk classes all form part of the model. The only purpose of these classes are to store the sun data and tree data in a meaningful way and to provide access to the data.

The controllers are made up of the Simulator and SunThread classes. These classes use user interactions from the GUI to determine how the should work and they are the only classes that have the ability to change the model of the program (ie. Use the setter methods in the Land, Tree and LandChunk classes).

The view part of the program is made up of the ForestPanel and TreeGrow classes. These classes are not able to change any of the data in the model (they cant use the setters method from classes in the model) but they do use the data in the model to update the GUI and so they only have access to the getter methods of the classes in the model.

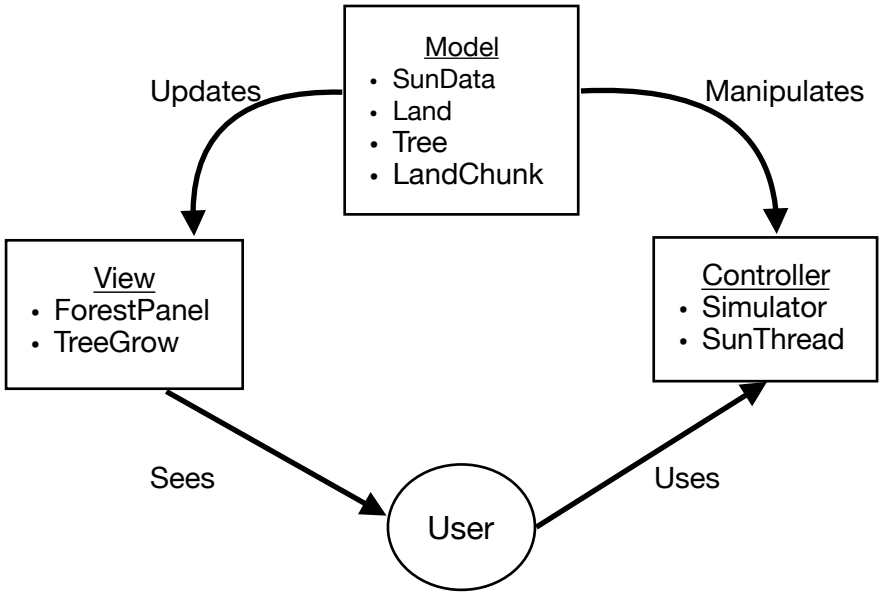

Figure 8: My programs MVP architecture

Further investigation

For further investigation, I measured the time taken to run the program when different sizes of LandChunk block were used. Figure 9 below shows a comparison of the time taken to process the trees for each year given block sizes of 3000x3000, 1000x1000, 100x100 and 50x50.

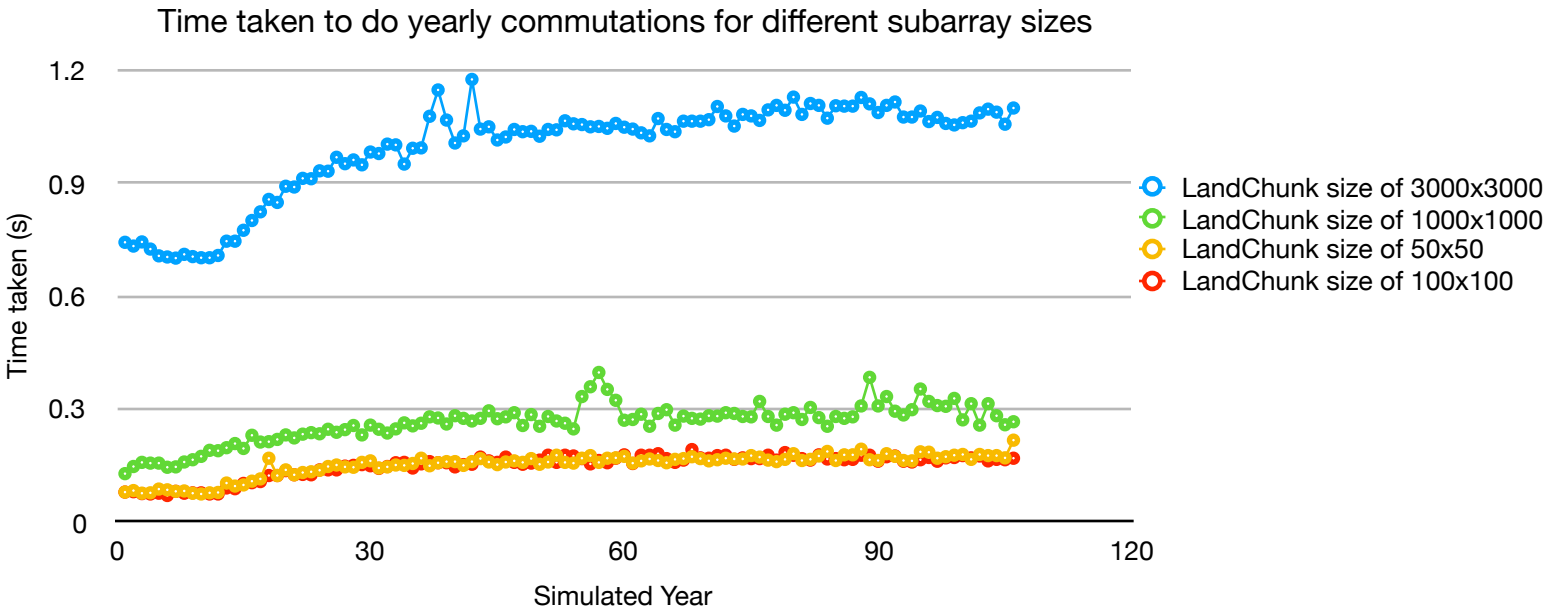

Figure 9

In the case of the 3000x3000 block, it would mean that for each tree that is processed, the entire sun-map dataset would need to be locked and so essentially the program would be sequential because no two threads would be able to access and update the sun-map dataset at the same time. As the block sizes are decreased, more and more threads can be processed simultaneously and so, as seen in Figure 9, the time taken for the each year to process decreases as the clock sizes become smaller. This shows that the program can still be efficient even after concurrency has been implemented.

I then wanted to investigate the difference in simulation time using the same block sizes but with and without using any locks. Figure 10 below shot the results from this test. It is evident from the graph that using locks does increase the simulation time of the program but only by a very small amount. This is the trade off that must be accepted given that using locks will make the data more accurate. The graph also shows the time taken to do all the calculations when the program is running completely sequentially and so it can be seen that using locks on the parallel program still provides speed up.

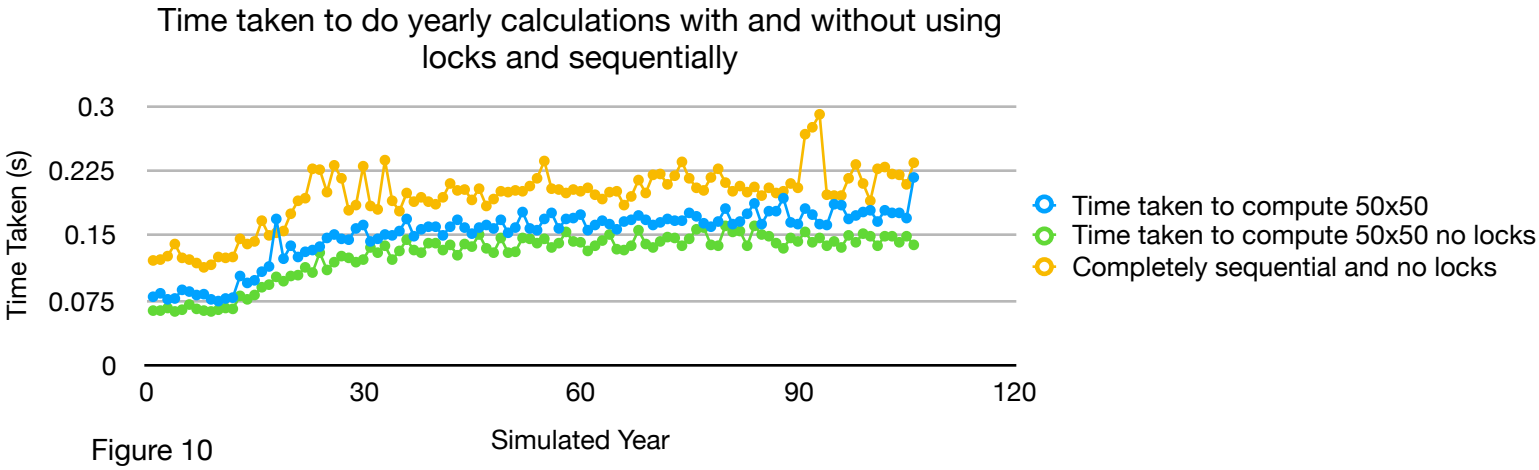

Figure 10

Conclusion

The program has now been made in such a way so that different Trees can be processed in parallel while using concurrency measures to ensure the integrity of the data. Processing the trees in parallel improved the performance of the program in a more significant way compared to how much the performance is decreased due to using locks as was shown in Figure 10. And so it is still worth it to use parallelism for the program.

Various steps were taken to ensure that thread safety was provided to all shared variables in the program. Thread synchronization was used to block certain parts of the sun-map dataset and a universal order of locking was used to ensure that no deadlocking would occur. I also used a small sample set to demonstrate that locks prevent race conditions from occurring.