Final Project:
Simulation of Fire Propagation in Java

Clint Olsen and Chase Heck

ECEN 4313/CSCI 4830: Concurrent Programming

May 1, 2017

University of Colorado
Boulder

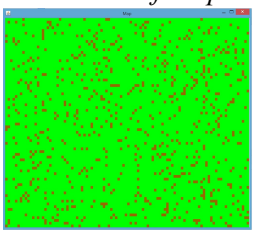
Problem Statement

For our project we present an investigation into the efficacy of simulating the spread of forest fires using concurrency. Present models of this phenomenon are complex and difficult to model mathematically with any consistency. Instead, we opted to simulate the propagation of a forest fire utilizing multiple threads that concurrently compute the probability of trees catching fire as they spread. This problem inherently lends itself to a multithreaded approach due to fire naturally propagating in many directions at once. We discovered that linearly, such a problem does not handle efficiently due to the sheer amount of options that a fire has in which to spread. Furthermore, a linear solution does not accurately display how a fire might spread in multiple directions simultaneously; instead tending to snake outward in individual tendrils. The sheer amount of probabilistic computation that needs to occur in parallel was one of our biggest bottlenecks, and we opted to utilize two different concurrency schemes to benchmark an optimal solution.

Problem Solution: Continuous Thread Generation

The simulation is modelled using a 2D matrix, with a skeleton merely printing an array comprised of indices that are either "trees" or "spaces." We utilize a 100 by 100 matrix that determines whether the cells are trees or not based on probability. This way, each "map" of our forest is dynamic, but by modifying the probability we can simulate different forest types. In doing so, we can see how the fire would propagate in a dense or sparse forest. The user has the option to determine the density as well as wind direction and humidity (covered later) upon launch of the program. The array is physically linked upon creation, with pointers to the logical up, down, left, and right surrounding nodes (denoted by their corresponding topographical directions). Nodes that are on the edge of the array point to null values that when accessed will stop the fire propagation to avoid null-pointer exceptions. Once the map variables have been determined, a Map object that stores all user input data and starts the fire is created. The output of the array is mapped to a JFrame screen that allocates color coded cells to the indices of the array. Green indicates the presence of a tree, while brown indicates the lack of a tree. The user can click anywhere on the screen to begin the propagation.

Figure 1: *Graphical Representation of Map*



Once a fire has begun to propagate, the map object initiates a main thread that loops four times; once for each possible direction (north, south east west). Upon deciding on a direction, the thread determines whether or not the cell in that direction should catch on fire probabilistically. This initial probability is contingent upon our "humidity" variable, which is passed as either "low," "mid," or "high" by the user. Humidity influences the probability of fire spread by simulating how a forest might potentially burn were it dry or wet in a given scenario. The higher the humidity, the lower the overall probability of a tree catching fire. Probability is calculated by generating a random number from 0 to 200, and stipulating that said randomly generated number must be greater than a given value in order to proceed. For example, at a low humidity the RNG need merely be greater than 30 to spread the fire, but at a higher humidity the probability could be diminished to greater than 100 (50-50 chance). This metric is certainly a little arbitrary, but gives a good general idea of propagation.

```
if (n > humidity) {
        current.north.onFire = true;
        current.north.c  = new Color(255,0,0); //Orange red for fire
```

Once a cell has been determined and the humidity probability has passed, the main thread sets the onFire value of that indice to be TRUE and then generates a new thread. This new thread is passed the same map object used by the main thread, but with the newly burned cell as "on fire" and the propagation point of the thread. This thread then repeats the process described above with the new map version.

```
//creates a new thread to start burning at this spot

ThreadNode tn = new ThreadNode(current.north, this);
tn.start();
```

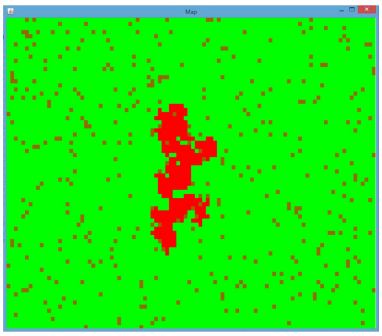


Figure 2: Start of propagation

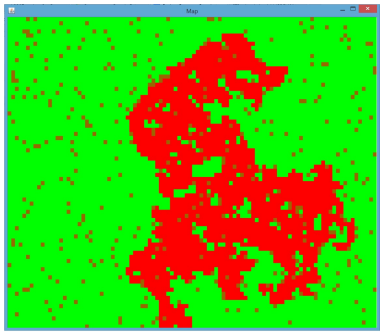


Figure 3: Progression of fire over time

As the program runs, more and more threads are generated which consequently step through their own for loops. At every direction each thread calculates the probability of the fire spreading and generates a new thread (or doesn't) which repeats the process starting at that node. Aside from humidity probability, each thread must also check that the new direction is not the edge of the map (current.next == null), that said direction is actually a tree (if it is a non-tree cell it must break), if the cell is already on fire, or if the cell is the direction the thread came from. Initially we experienced some problems with fires doubling back on themselves, which is a waste of resources and inherently unrealistic. Furthermore, we determined it to be a waste of threads to burn and propagate through threads that had already been set alight. Once each thread has completed its loop over the four possibilities, it gives up the processor, allowing the propagating threads to continue.

To add a layer of probabilistic complexity we incorporated a wind simulation to our fire propagation. The algorithm is comprised of an additional level of random number generation. The user is prompted to supply a wind direction at runtime, and said direction influences the probability of spread in much the same way as humidity. If a thread steps through its for loop and attempts to go a certain direction, it first checks if the direction it is heading is the same as the direction of the wind. If so, the probability remains unchanged (it is extremely likely the fire will spread). If the wind is blowing perpendicular to the direction of spread than the probability is diminished a fair amount. And if the wind is blowing in the opposite direction the probability is effectively halved.

```
else if (wind == 's') {
        if (n > probBurnOp + humidity) {
        if(current.north.l.tryLock()){
                try{
                        current.north.onFire = true;
                        current.north.c  = new Color(255,0,0);
```

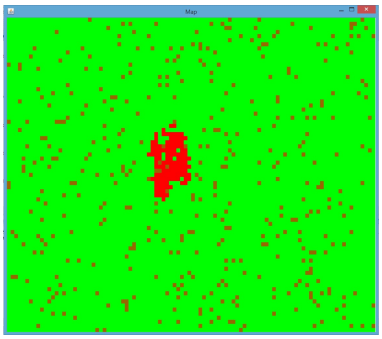


Figure 4: Start of propagation with wind

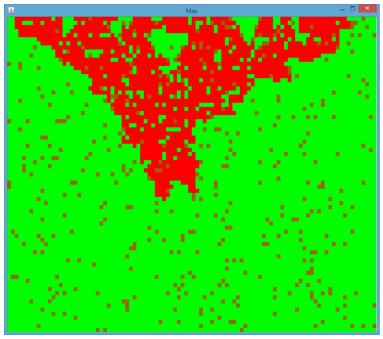


Figure 5: Progression of fire over time with northward wind

This yielded some very interesting results, especially when compounded with different humidities and tree densities.

Concurrency Issues

We identified three areas that necessitated mutual exclusion of our threads. This problem is unique in that there is little reason to lock index of the map from being burned if they have already being burned by another thread. However, trouble arises when threads attempt to burn a cell simultaneously. Though the program will run normally in this situation, it is a huge waste of resources and ultimately not very realistic. Once one thread determines it can burn an adjacent cell, it locks out all other threads until it had finished updating the onFire value of that cell and updating the map. This way, other threads attempting simultaneous access will be forced to move on. The tryLock() method is used to allow threads to "bounce off" of cells that are currently being burned and potentially move on to another direction. This improves not only the time of the program, but the realism of our probabilities.

```
if(current.north.l.tryLock()){
...
}
else{
//do nothing and move on
}
```

We also found it necessary to have a nested lock when the thread updates the array/map and the subsequent JFrame. If threads do not lock here the map tends to break and update cells excessively, occasionally leading to null pointer exceptions and incorrect visual representation.

```
//print map on graphics display

l.lock();
try{
        frame.invalidate();
        frame.validate();
        frame.repaint();
        }finally{
                l.unlock();
        }
```

Finally, in order to benchmark our program, we found it necessary to take both the runtime and the number of threads generated by each execution. To do so we incremented a total runtime variable by the runtime of each individual thread. Obviously, the static total runtime variable and the thread counter needed to be mutually exclusive.

```
timeLock.lock();
try {
        total += (System.currentTimeMillis() - last);
        last = System.currentTimeMillis();
        counter++;
        }finally {timeLock.unlock();}
```

Alternative Solution: Thread Pool

Finally, to benchmark our project we opted to compare our thread-spawning algorithm to one that implemented a thread pool. It is quite obvious that a linear implementation of this problem would be infeasible, so we forewent it entirely. The thread pool variant essentially created a request for a new task (startPropagation method) any time that a thread determined the fire should be spreading and a new thread should be spawned.

```
//creates a new worker thread to add to executor service and start
   //burning at this spot
WorkerThread t = new WorkerThread(current.north, this);
MapBuildTest.executor.execute(t);
```

Instead of generating these threads each time, these tasks were submitted to an executor service which used a fixed thread pool to divide up the workload among the set number of threads.

```
//create the executor service and thread pool
public static ExecutorService executor =
   Executors.newFixedThreadPool(ThreadPoolSize);
```

With this implementation, we were able to play around with the number of threads in our pool to determine an optimal run (70 threads appeared to be the high water mark for time improvement). This number was determined to be optimal because the speedup and accuracy of the propagation began to level out after this point. See experimental results below.

Experimental Results

In comparing the two main solutions to this problem, thread pools or thread spawning, there is a drastic difference in terms of performance. Looking at the thread spawning approach, the program must allocate memory on the fly for new threads as they are needed. In situations where the fire has high associated probabilities, a number of threads as large as the area of the map may be generated. On the other hand, a thread pool can effectively complete the same number of tasks, but by reusing a set number of threads. The benchmarked case is the scenario of high tree density, no wind, and low humidity.

The thread pool technique results in a dramatic speed optimization, nearly doubling the speed of the thread spawner. The graph shows the number of calls to the propagation method in the thread pool case, and the number of threads spawned in the other case. These numbers represent the the amount of work in both and are used to compare the time vs. work of each algorithm. The second case that was tested added two additional parameters, a northward wind and medium humidity level.

The same trend is recognized in this case as well, which shows that the probabilistic determined by the wind and humidity factors is consistent. This also shows the work reduction (fire reduction) resulting from an increase in humidity and more directional propagation. The final case tested is low tree density, no wind and low humidity, to show a contrast in work from the other extreme of high density, no wind, and low humidity.

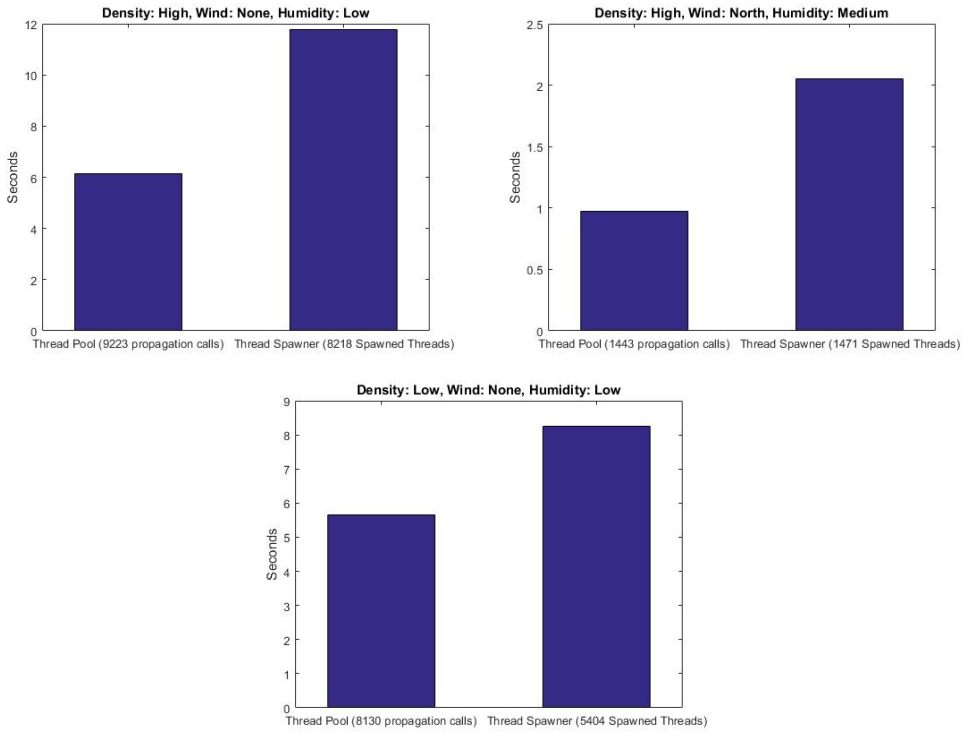


Figure 6: Benchmark Result 1, Benchmark Result 2, Benchmark Result 3

In general, the scenarios that include no wind and low humidity have discrepancies in workload. The thread pool seemingly displays a more realistic outcome for the fire by eventually burning the whole map,

which makes sense due to the low antagonistic impact of wind and humidity on propagation. The thread spawning technique, however, tends to burn the majority of the map, but not to the same extent, explaining the lower workload. The cause of this is likely related to the sheer number of threads that are in existence in the spawner at a given time. This large number of threads as the fire propagates becomes much larger than the thread pool size, so the program is trying to do more at once. This causes more contention between threads and a higher risk of threads attempting burns in directions that are locked by other nodes, and exiting because the thread exhausts all possible moves. Overall, the thread pool method was an outstanding optimization to this problem, making the solution much more practical, especially on a large scale.

Conclusion

Overall, this project was very insightful in regards to many aspects. The first was the realization that some problems are inherently parallel, such as this project, and would be very difficult to implement linearly. Upon looking into comparing this implementation with a single threaded program, it became obvious that the single threaded approach would not be useful, practical, or correct. The next aspect was the complex nature of using probability to drive a simulation. We were able to implement 2 probabilistic components, which was a large majority of the work done on the project. Upon further research on this topic, however, the number of parameters, probabilities, and randomness that goes into a full blown fire spreading simulation is staggering and would likely require a large cluster or supercomputer to produce meaningful results. A final realization that was made was the significant impact of thread manipulation techniques on speed optimization. Upon switching from the thread spawning version of the program to the fixed thread pool, there was nearly a doubling of speed, which gave a tangible representation of the potentially antagonistic effects of memory allocation and object construction.