Assignment 6: Concurrency in Java
CSC 254

Emily Michel and Sarah Jeter

December 14, 2016

Figure 1: It's an Ex*egg*utor! Get it?

1 The Basics

1.1 Running our Code

Our version of the code is in two separate folders. Each version of the code is in a separate folder with a copy of the coordinator. To compile, while in the directory that contains the Executor and Thread folders, use the command `Executor\javac *.java` to compile all executor files and then `Threads\javac *.java` to compile the thread code. The code is then run using a version of the following for the executor:

```
java ExecutorLife --glider -s 100000 -t 5
```

and the following for the threads version:

```
java ThreadLife --glider -s 100000 -t 5
```

1.2 Playing Conway's game of Life

In this game, the board is a two dimensional array of cells, each of which can potentially be a 1 or a 0, where a 1 means there is a dot in the cell. To determine if a dot survives to the next generation, we check each of its neighbors including diagonals (a total of 8). If it has 0, 1, 4, 5, 6, 7, or 8 neighbors, it dies, 2 or 3 lets the dot survive to the next generation, and if there are exactly 3 neighbors, then a new dot is generated in that cell. In Figure 2 a glider is placed in the two dimensional array A. When `doGeneration()` is called, the method checks each space around that cell to determine its number of neighbors and updates the second board accordingly. This second board is the next generation and it does not switch to be the current generation until all threads are at the barrier. In the second board in Figure 2 the blue circles are the ones that survived to the next generation and the black dots are new. `A[1][0]` has exactly three neighbors `A[0][1]`, `A[2][0]`, and `A[2][1]`, so a new dot is placed at `B[1][0]`.

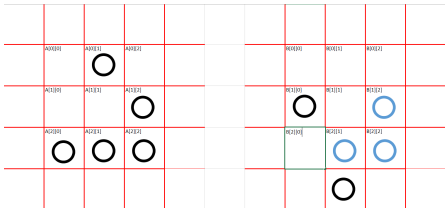

Figure 2: Example of 1 generation of a glider

1.3 Division of Labor

For this project, Sarah worked on the threads section and Emily created the executor.

2 Threads

By creating threads for each section of the board, we are able to execute the calculations in parallel allowing each generation to finish faster. When the code is run, we specify the number of threads we want with `-t`. This then divides the LifeBoard into `v` pieces and calculates the new board. After all the threads have finished one generation, we repaint the LifeBoard to show the updated board.

In order to ensure that a new generation does not begin before every thread completes the previous generation, we use the `CyclicBarrier` java class. This class has a function called `barrier.await()` which waits for `N` threads to be waiting upon the barrier and then the barrier breaks. The barrier also gives the option to execute a section of code automatically after the barrier breaks. This is done by creating a new Runnable and placing the code you wish to execute inside the run method. In our case, this is where we switch the boards and repaint the Lifeboard, or if running with the `--headless` function, the generation variable is incremented. After that code is executed, the threads begin executing the next generation.

3 Executor

Instead of directly creating and using threads, I used an `ExecutorService` from the `java.util.concurrentlibrary`. `ExecutorService`s take `Runnable` or `Callable` tasks and run them on a certain number of threads. The service is responsible for assigning tasks to threads. If it is given more tasks than threads, it will execute the excess tasks as threads become available.

`OnRunClick` creates `ExecutorService` and calls the helper method `generate Tasks` to produce several `Tasks` that manipulate the board. Task, a modified version of `Worker`, is a Runnable, which can be converted to a Callable and passed to the ExecutorService. The run method of each task is responsible for performing part of a generation on a given section of the board. Tasks are created using the helper method `generateTasks(k)` where `k` is the number of tasks (more on the number of tasks in the **Testing** section). Tasks are assigned a section of the board using the same process as in the Thread implementation. The run method of Task is similar to that of Worker, except that it does not have a while loop and only calls `doGeneration` once. This is necessary to properly synchronize the threads. For a given generation, we need all threads to update part of a version of the board, and only once they are all finished can we set the board to this version and finish the generation, otherwise the updates could interfere with each other.

`ExecutorService`'s `invokeAll(...)` method is used to ensure that every Task is completed before the board is saved and the generation is updated. `invokeAll(..)` takes a Collection of Tasks that will manipulate the board and executes them, only returning after every task has completed. Once it returns, it is now okay to save the board. This is done by calling method `finishGeneration()`, which saves the board that the Tasks have just updated.

It then updates the generation if headless is being used or repaints the board. `invokeAll(...)` and `finishGeneration()` are in a `while(true)` loop, which will repeat the process for multiple generations. The process is summarized in Figure 3

You'll notice that this loop is in a Runnable called `TaskController`. Originally, I put it inside method `OnRunClick`. This works with the headless version, however it is a problem with glider. Due to the nature of Java graphics, `repaint()` must be called by the main thread. With the `while(true)` loop inside `OnRunClick`, the main thread is kept busy and the JPanel is not able to repaint itself. Putting this loop in `TaskController` lets it run on a separate thread, freeing the main thread to deal with graphics. So `OnRunClick` creates an instance of `TaskController` and assigns it to the executor. The `ExecutorService` and the list of Tasks are passed to `TaskController`. `TaskController` is run on a new thread and uses the `ExecutorService` to handle the execution of the list of Tasks, as previously described. This leaves the main thread free to handle graphics when repaint is called. Note that only one `ExecutorService` is being used for `TaskController` and the Tasks that update the board. Thus, it needs one more thread than specified by the user as the user specifies the number of threads that manipulate the board. For example, the call:

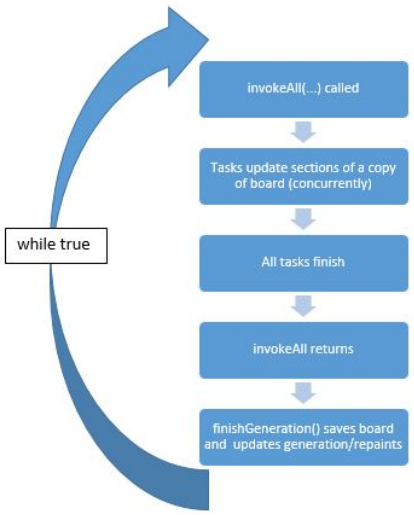

Figure 3: Process of while loop where each iteration is one generation

```
java Life --glider -s 100000 -t 10
```

will create an `ExecutorService` with 11 threads: 10 for manipulating the board and 1 for the `TaskController`.

4 Testing

In testing each of our methods of concurrency, we used

```
java Life --glider --headless -s 100000 -t 2
```

to run our code, changing the value of `-t` each time. The `--headless` argument prints the system time every 10 generations. This allows us to correctly time the speed up as we add more threads. Simply using

```
ssh node2x14a.csug.rochester.edu -Y
```

will not give us very accurate results by just timing the glider on the screen because it has to forward the graphics to whatever computer we are sitting at. We used a `-s` of 100,000 so that the generations were slow enough to see with the human eye. In order to replicate our results, this parameter must be used.

4.1 Threads

In the testing of the code modified to use threads, we tested up to 56 threads which is the maximum possible on the `node2x14a` machine. As can be seen from Figure 4, the more threads we use in running the program the faster the execution time is. We that the program runs faster each time until about 12 threads are running. At this point we see an increase in the execution time. At 24 threads this time starts to decrease again; however, even with 56 threads, the maximum possible on this machine, we still get a faster execution time with 12 threads. This is further shown in the Figure 6.

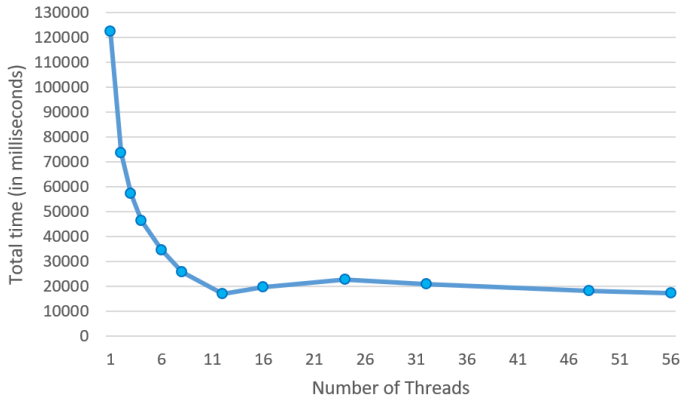

Figure 4: Total time to execute 400 generations with threads and 100,000 spin cycles

4.2 Executor

Since Executors deal with assigning tasks to threads, it is acceptable to give an executor more tasks than threads. If there are more tasks than available threads, the executor will assign them once a thread opens up. In some cases, it might be better to have more tasks than threads. The more tasks there are, the shorter each task is, but less tasks can be executed at once because the executor needs to wait for an available thread. I experimented with the number of tasks on the cycle servers. Increasing the number of tasks improved performance, and it peaked before the number of tasks was twice the number of threads.

Therefore, I decided to have the number of tasks be 1.5 times the number of threads.

The graph in Figure 5 shows the total time (in milliseconds) of 400 generations vs the number of threads. As expected, the time decreases as the number of threads increases, more drastically at first and then leveling off around 12 threads. This makes sense because as the number of threads increases, so does the amount of overhead in the executor.

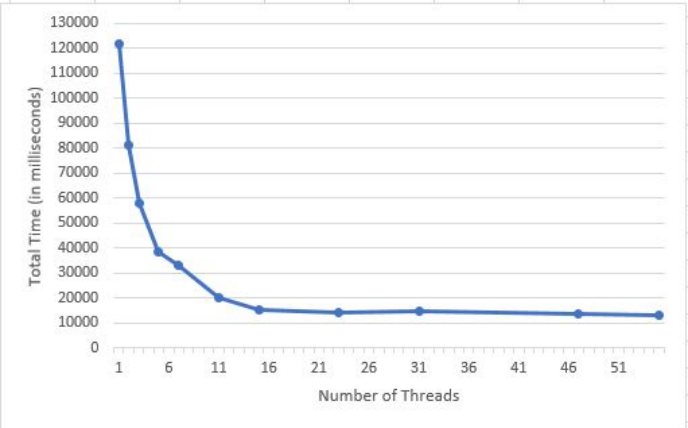

Figure 5: Total time to execute 400 generations with executor and 100,000 spin cycles

4.3 Speed Up

Each point on Figure 6 is calculated by dividing the time it takes for the original code to run 400 generations by each data point in Figures 4 and 5. The peak is at 12 and the program does not reclaim this speed until 56 threads. Reasons why this happens are discussed later in Problems and Improvements. In the prompt for the assignment, it stated that "Ideally, you'd see a speedup of k with k threads." As can be seen in Figure 6, the ideal speed up is matched up to about 4 threads. Here the calculated speed up slows down and then after about 12 threads it levels off.

This can be due to a multitude of things. In the threads version, the cyclic barrier could be slowing things down. Because it causes the threads to wait until all threads are at the barrier, some threads that are faster have to wait for the slower threads. As you increase the number of threads, you have to wait for more threads to get to the barrier. Similarly the executor version is also worse than the ideal due to the overhead of the executor handling the threads. The amount of tasks may also be a factor here, tweaking the number of tasks in could potentially improve speed up. At first, the executor version is slower than the thread version, but then it becomes faster. This could again relate to

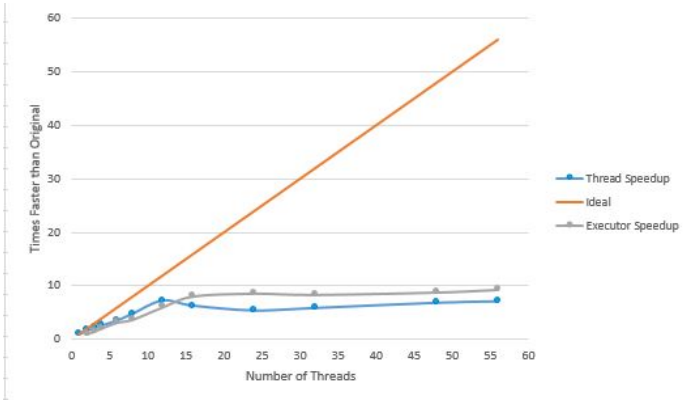

Figure 6: Calculated speed up of threads vs. ideal speed up

the overhead of executors vs. the overhead of threads. The number of tasks may also become a better optimization as the number of threads increases.

5 Problems and Improvements

5.1 Stop Button

One of the issues we noticed is with the stop button on the JPanel. It will work at times; however, sometimes when clicked, it will freeze the whole JPanel and you have to terminate the program. We did not have a chance to fix this and other teams seemed to be having the same issue.

5.2 Executor with One Thread

Executor does not have any data for 1 thread. When I ran it on the node2x14 server, it did not work and appeared to start an infinite loop of print statements but was not actually making generations. I ran it on the cycle servers and the same thing happened. Later I tried to run 1 thread on the cycle servers again and it worked fine, so I might have typed something wrong the first few times. There wasn't enough time to test/debug it on node2x14 so it may or may not work for you.

5.3 Number of Tasks on node2x14

I chose to have number of tasks (K) equal to 1.5 times the number of threads (T) after experimenting on the cycle servers. I assumed this optimization would also help on the node2x14 server. I tested with K = T and K = 1.5T on node2x14 with thread sizes 12 and 24, and 1.5T actually seemed to have worse performance. It might have something to do with the large thread sizes or the

machine. There was not time to explore it further. I think maybe 1.5T might not be the best and I need some other multiple of T between 1 and 2.