CSC 254: Concurrency Assignment Cellular Automaton / Game of Life

Abdullah Al Mamun and Anup Pokhrel

December 15, 2016

Abstract

In this assignment we experimented with Threads in Java. We parallelized an existing implementation of Conway's Game of Life. This is also called *Cellular Automaton*. By default there's a few organisms, we can change the organisms by running the GUI where we can turn on/off dots in the board. The organisms moves through a series of generations. The rules are, if an organism has two or three neighbors (counting diagonals), it survives to the next generation. If it has fewer than two neighbors it dies of loneliness. If it has more than three neighbors it dies of overcrowding. If an empty cell has exactly three neighbors a new organism is born in that cell in the next generation.

In this assignment we experiment with two types of threads implementation. One version uses threads directly, standard in Java 2, and the other version uses the Executor facilities in Java 5/6/7/8.

Files Included

- executor
  -Life.java
  - Coordinator.java
- threads
  - Life.java
  - Coordinator.java
- original
  - Life.java
  - Coordinator.java

How to Run

Go into a directory and run the command below, assuming that the grader will be trying to run it through a terminal setting. This will compile the files.

$javac\ *.java$

If the grader were to use, an IDE suck as Eclipse then simply import this whole project and run it from there.
Inputs
For the files inside the *threads* directory, the way to run it after compiling through the previous mentioned step is to run,

$java\ Life\ -s < number > -t < number >$

Here $-s$ takes in the number of spin and $-t$ takes in the number of threads the user wants to input.

In our testing phase we have observed that 400 spin gives us good understanding of how the threads are working.

For the files inside the *executor* directory, compiling is again the same but to run it the user has to input,

$$java\ Life -s < number > -t < number > -k < number >$$

Here $-s$ and $-t$ takes in the same parameters as before, in addition $-k$ takes in the number of tasks the user wants to divide the whole task into. We want to note that, $-k$ parameter can be optional. So if the user doesn't input anything for the k parameter then the number of tasks will be assigned same as the number of threads. The user can also mention the parameter and give the same number of tasks as the threads, manually.

Implementation Details

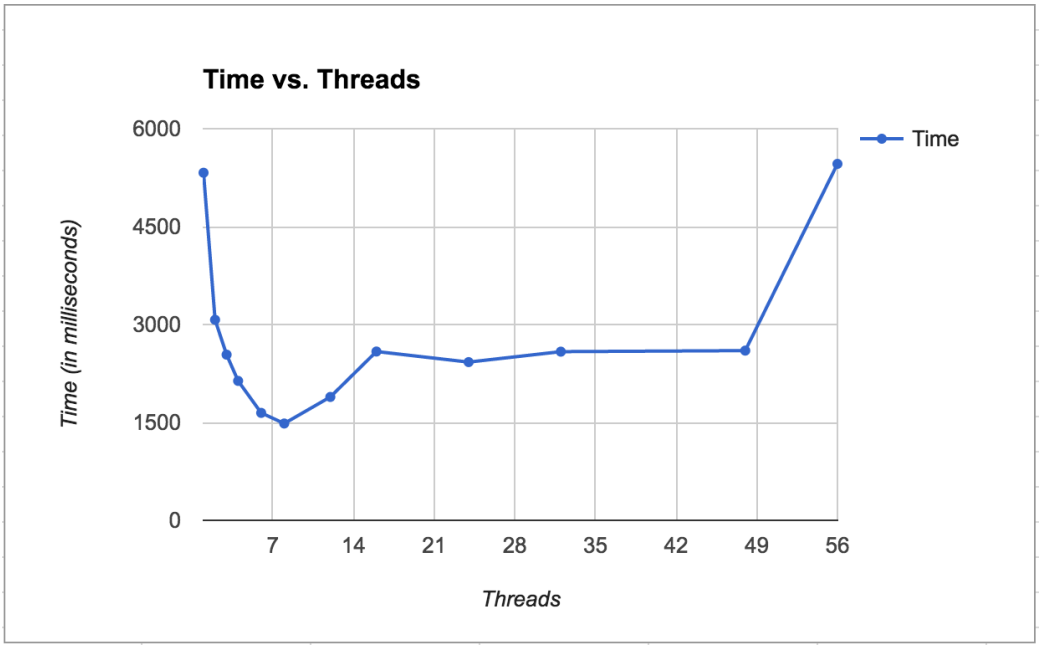
In the threads version, we decided to use Cyclic Barrier from the *java.util.concurrent.CyclicBarrier* instead of using *join()*, *notifyAll()* and other such built in methods from Threads, as that would have complicated maintaining the threads. We have also made sure that the changes correctly reflect in the GUI, so when we use threads we can see how the process is speeding up.

In the executor version, we have used *java.util.concurrent.Callable*, *java.util.concurrent.ExecutorService* and *java.util.concurrent.Executors*. So in this executor version what happens is, first the specified number of threads gets created. Then, the board gets divided into specified number of tasks, if not specified then same number

of tasks as the threads, and then these tasks are divided among the threads. Good thing about the Executors is that, we don't have to worry about which thread is going to take care of which tasks, it's getting taken care of internally. For the generation control, we use a method called *invokeAll()* which temporarily halts the threads going into a next generation before all the threads have completed their assigned tasks.

Speed Up Analysis

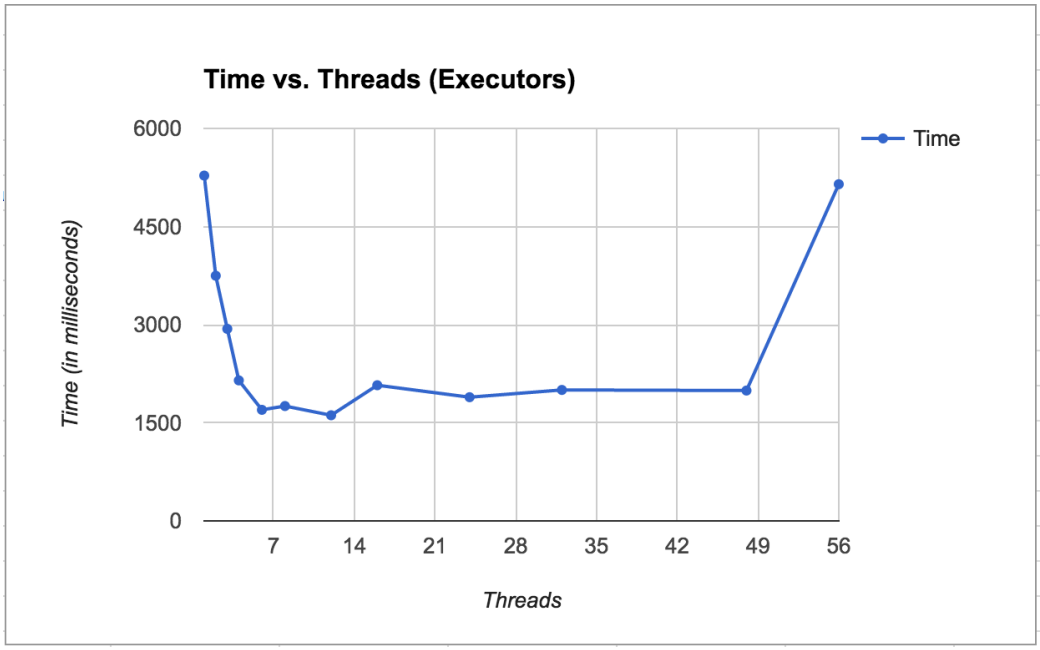
We have tested our code in the cluster in node2x14a for performance purposes. For the spin we had used 4000, and thread number 1 represents the runtime for original unmodified code. Below are the graphs for performances and the relevant data tables are right below them.



Threads	Time
1	5325
2	3072
3	2538
4	2136
6	1646
8	1483
12	1890
16	2586
24	2424
32	2584
48	2600
56	5461

Table 1: Using threads directly

From the graph and the table above we can see that, for 1 thread it takes a long time and it decreases as we increase thread number, but again after 8 threads it starts to increase.

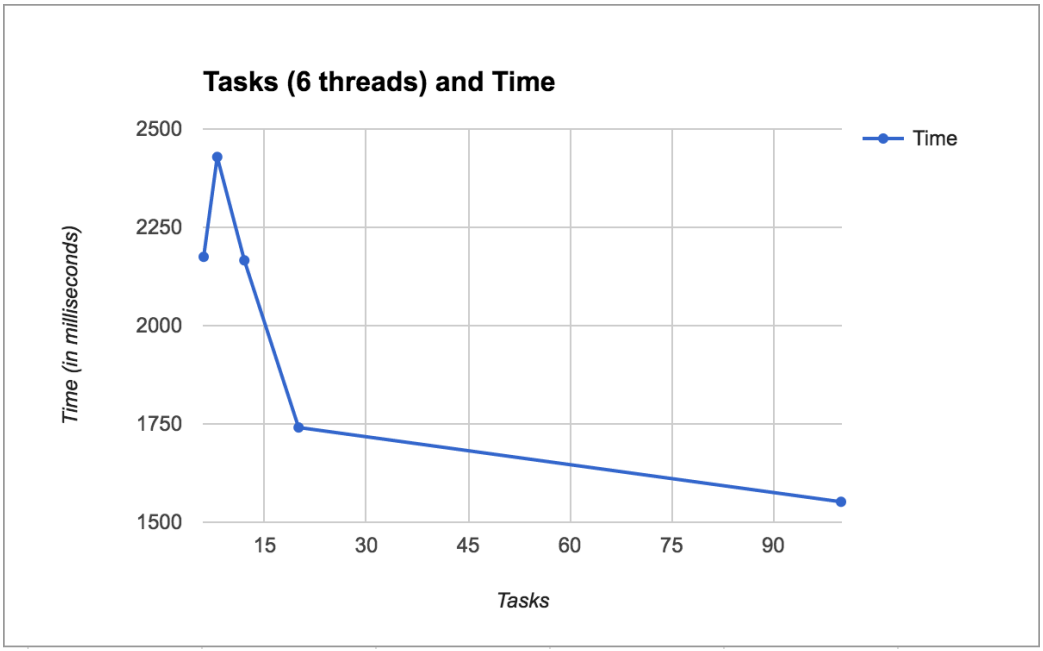


Threads	Time
1	5279
2	3748
3	2936
4	2148
6	1698
8	1756
12	1615
16	2074
24	1891
32	2002
48	1993
56	5147

Table 2: Using Executor

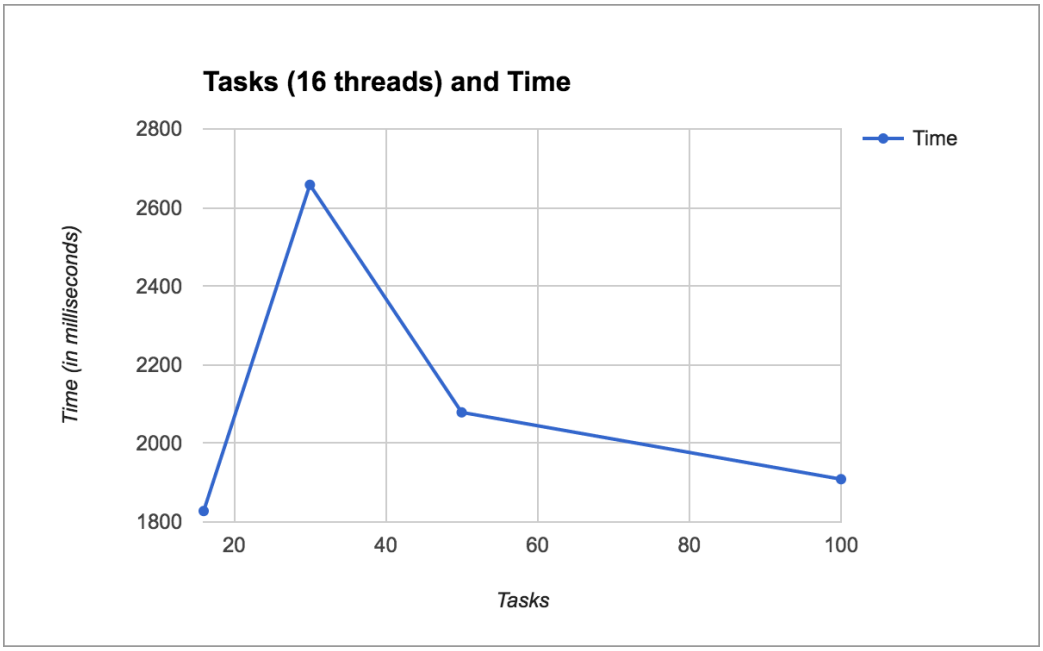
From the graph and the table above we can see that, for 1 thread with executors and same number of tasks as thread it takes a long time and it decreases as we increase thread number,

the there's a sudden spike with 8 threads, then it keeps going up and down. What is interesting here is that, for 48 threads it the executor version does considerably better.



Tasks(6 threads)	Time
6	2175
8	2429
12	2166
20	1741
100	1552

Table 3: Using Executor with 6 threads and different number of tasks



Tasks(16 threads)	Time
16	1828
30	2658
50	2079
100	1909

Table 4: Using Executor with 16 threads and different number of tasks

From both graphs and tables above (Table 3 and 4) we can see that, after a certain number of task increase the time to run the program comes down considerably.

Extra Credit

We have implemented the 1st extra credit that is, adding a button "Step" that will step the application exactly one generation.