Analysis of Conway's Game of Life with OpenMP Implementation.

John Sabo
Parallel and Concurrent Programming
Rowan University

March 15 2024

Abstract

This paper presents an analysis of Conway's Game of Life implemented with OpenMP parallelization. The program, written in C, utilizes ten functions to simulate the game. OpenMP implementation parallelizes the main loops, significantly enhancing performance. Verification tests ensure output consistency between sequential and parallel versions. Performance evaluations reveal substantial speedup for larger matrix sizes with increased thread count, while smaller matrices exhibit minimal improvement due to overhead. Recommendations include testing larger matrix sizes for better performance gains.

I. Program Design

The Conway's Game of Life script is written in the C Programming Language and contains ten functions. Firstly, the 'getTime' function is used to get the system time in seconds. This is used to calculate the run time of the program. Secondly, the 'allocArray' function is used to allocate memory for the two arrays that will be used in the program. One array will contain the current generation while the other will contain the previous generation. Initially the previous generation is an array filled with all zeros. Thirdly, the 'initArray' function is used to initialize an array with all of the same specified value. It is used to populate the inital previous array. Fourth, the 'initTestArray' is used to test the validity of the multi-threaded version of the program. The 'test' script will run two versions of Conway's Game of Life, one with OpenMP and one without. It will then use the 'diff' command to verify that the two programs produce the same output. Fifth, the 'printArray' is used specifically for debug and testing purposes. It is used in the 'test' script in the two versions of the program to output all boards created during a run that will then be verified with the 'diff' command. Sixth, the 'initRandomArray' function is used to populate the initial current array with all random values of either 1.0 or 0.0. Seventh, the 'ghostFill' function is used to copy the second and second to last column and row into the last and first column and row respectively. This allows for easier access when checking the neighbors in each cell of the grid through each generation. Eighth, the 'updateCell' function is used to individually update one cell. It is called N by M times for each generation and it calculates the sum of the neighbors and uses Conway's Game of Life original logic to determine the state of the current cell in the next generation. Nineth, the 'step' function is used to loop through every cell and call the 'updateCell' function. It also determines whether or not there was an update anywhere on the board in the current generation. Finally, the main function contains the initialization of all of the important variables such as the current and previous arrays, the N and M dimensions of the board, and the max generation and thread count. It also contains the main while loop that is used to run the program until there are either no updates on the board or the max generation has been reached. It switches back and fourth between the two arrays being current and or previous. In other words, after one generation, when the previous array has been overwritten with the next generation's data,

it now becomes the current array and the current array becomes the previous. The main function finishes by printing out the time taken for the program to run.

In order to implement OpenMP into this program, it was rather straight forward. I needed to make the three main loops ran in each genertation parallel. To do this I first needed to include 'omp.h' in the beginning of the script to gain access to the library. Setting up OpenMP on my own machine proved to be very difficult, thankfully Rowan's Elvis has it pre-programmed for its users. Next, I needed to specify in main how many threads OpenMP would be using throughout the program. I used a command line argument as the number of threads specified by the user. After this I added 'pragma omp parallel for' statements in front of all three loops. The first two being inside of 'ghostFill' and the third being the only loop inside of the 'step' function. Beyond just added this statement I needed to specify which variables would be shared, and which would remain private to each thread. For the shared variables inside of 'ghostFill', they were a,the current generation array, N, the N dimension of the N by M array, and M, the M dimension of the N by M array. For the private variables, they were i and j respective to which section of the array the for loop was responsible for i.e. i for the rows and j for the columns. They were made private because both variables where being incremented for each thread and the thread would need that variable in the state that it was provided to it. Inside of the 'step' function, the shared variables were the current and previous arrays, and the N and M dimensions. The private variables were again i and j, but this time also the boolean isAlive. This variable was used to perform an boolean or reduction across all of the threads. If any of the threads returned true the final state of the variable would be true. This allowed the program to determine if there was any update for this entire generation.

Instead of manually running the program over and over again for a variety of test cases and calculating averages in a spreadsheet, I opted to create a bash script that would run a series of commands a certain amount of iterations for each command and compute the average before then dumping it into a text file. In the first assignment, I was more naive and chose to calculate the averages myself leading to a lot more headache. Using a script to get my data proved to be the much less redundant method. Getting my data this way enabled me to write a python script to simply read the generated file with the averages for each command and plot the data using the matplotlib library.

II. Test Process and Planning

In order to verify that the output of the parallel version of the program was identical to the sequential version of the program I wrote the script 'test' in bash which compiled and ran a series of parameters against both the parallel and sequential versions. It would run a ten by ten grid for ten generations, followed by a one hundred by one hundred grid for one hundred generations, and finally followed by a one thousand by one thousand grid for one thousand generations. Each of these grid sizes and max generations was ran with one, two, four, eight, ten, sixteen, and twenty threads. Every single generation's matrix would be wrote into a separate text file for each version of the program. Once all the commands and been executed and the two text files were full of ones and zeros, I ran the 'diff' command against the two text files, comparing their content looking for any differences. If the two text files were identical, it would print out that they are and visa versa. Finally it finishes by cleaning up the extra text files and executables created utilizing the 'rm' command. Thankfully, the 'test' script worked out as planned and my parallel version of Conway's Game of Life was producing data that was identical to my sequential version.

In order to test the performance of the now verified parallel version of the program I wrote another script called 'runTimePerformance' that would run the script with the parameters five thousand by five thousand grid size for five thousand generations with one, two, four, eight, ten, sixteen, and twenty threads. It calculates the average run time of the program for each thread count over five iterations, essentially running the program thirty-five times in total. It then will write

each parameter set's average run time into a text file to be used to graph the data. I ran into an issue, however, where upon trying to execute this script on Rowan's Elvis, I was timed out before the script could complete. The script ran for three hours before terminating my connection. I decided to reduce the grid size to five hundred by five hundred and keep the generations the same. This greatly reduced the time it took for it to complete and I was able to run it on Elvis with no issues and produce some promising data.

In addition to determine the average run time of the program, the graduate students were asked to test a variety of matrix sizes and what their performance looks like. The matrix sizes included one by sixteen, two by eight, four by four, eight by two, and sixteen by one. In order to test the performance of these matrix sizes, I took a similar approach as I did to the previous script mentioned. The script named 'runSizePerformance' would do the same exact thing as 'runTimePerformance', however this time it would test each of the matrix sizes with each of the thread count, each for one hundred iterations, essentially running the parallel version of the program three thousand and five hundred times. This script has no issues running on Elvis. Just as 'runTimePerformance', 'runSizePerformance' would print all of its output to a text file to be used later to create graphs of each matrix size. The data produced for the size performance was interesting when compared to the time performance.

Visualizing the data produced would be straight forward with Python and the matplotlib library. I wrote a script first for the size performance script that would produce two graphs for each matrix size. One graph was the efficiency graph and displayed the time taken versus the thread count. Ideal we hope to see a negative trend in the graph. The second graph was the speed up graph which showed how many times faster the program was for each thread count when compared one thread, or the sequential version. Re-purposing the functions I created in this script for the time performance was a matter of copying and pasting and changing a few variable names.

III. Data

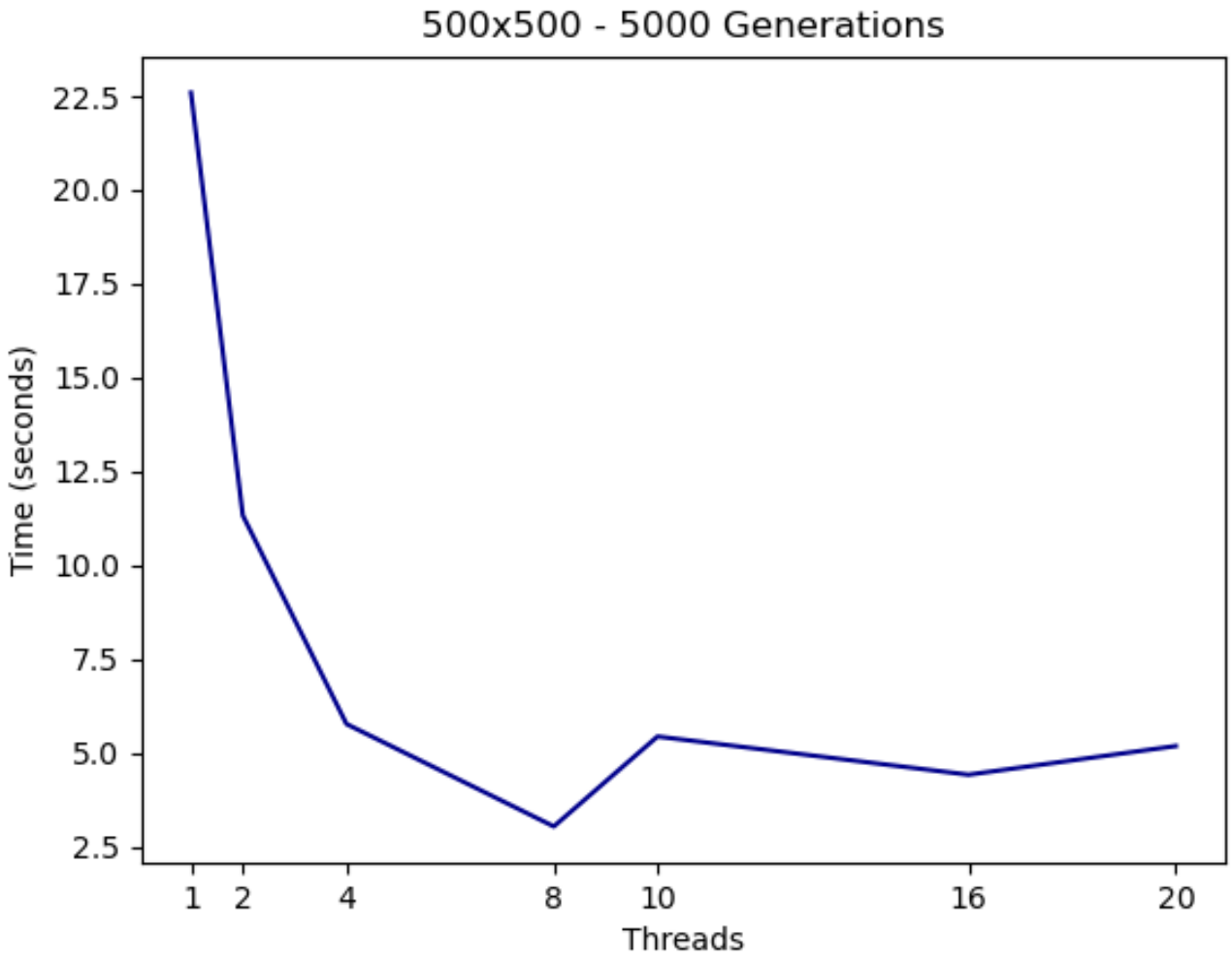


Figure 1: *This is the efficiency graph for the five hundred by five hundred grid ran for five thousand generations. Each thread count was ran for five iterations to determine its average value.*

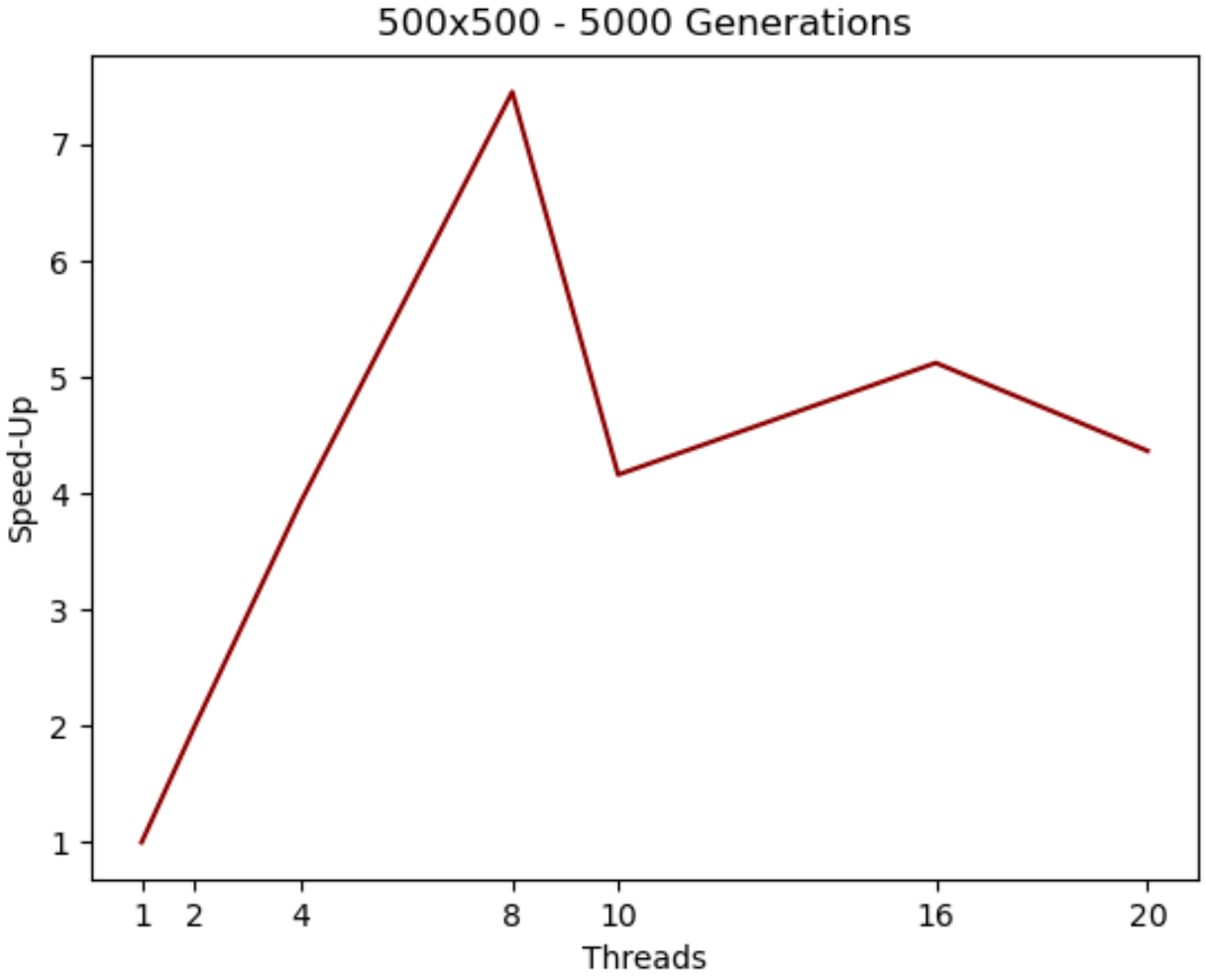
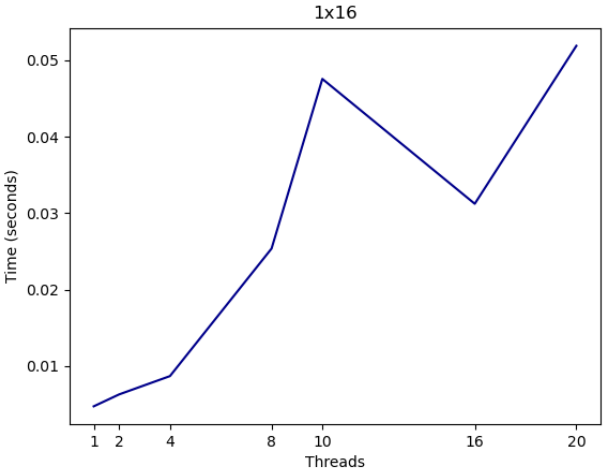
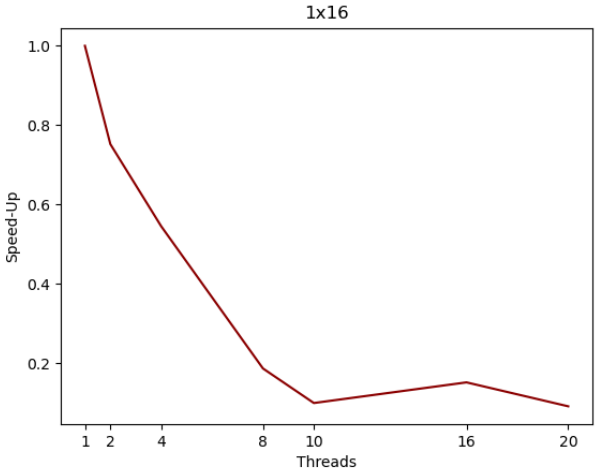


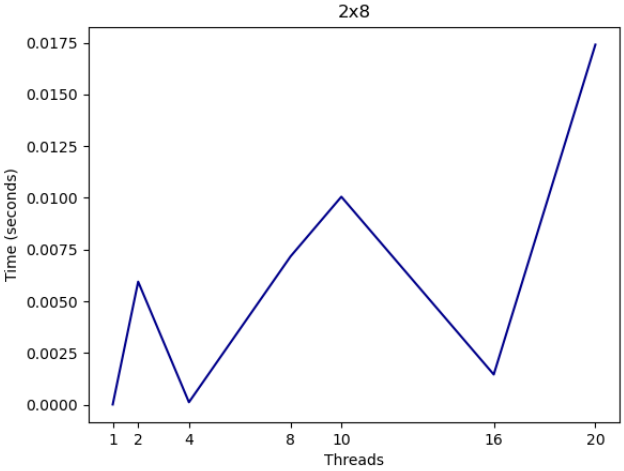
Figure 2: *This is the speed up graph for the five hundred by five hundred grid ran for five thousand generations. To determine each tread count's speed up, you divide the sequential time of the program by the time taken for that thread count. This is why one thread has one speed up.*



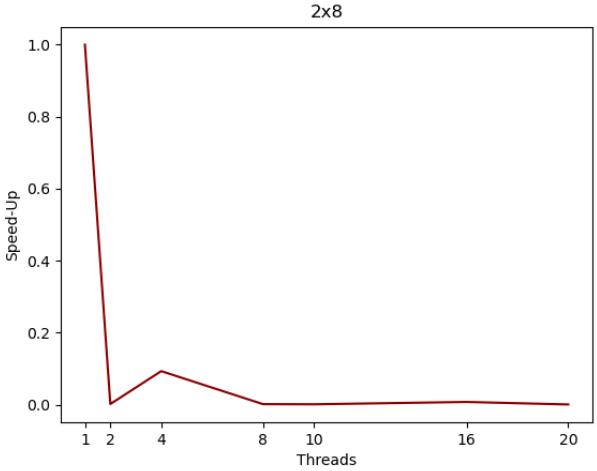
(a) *1x16 Efficiency Graph*



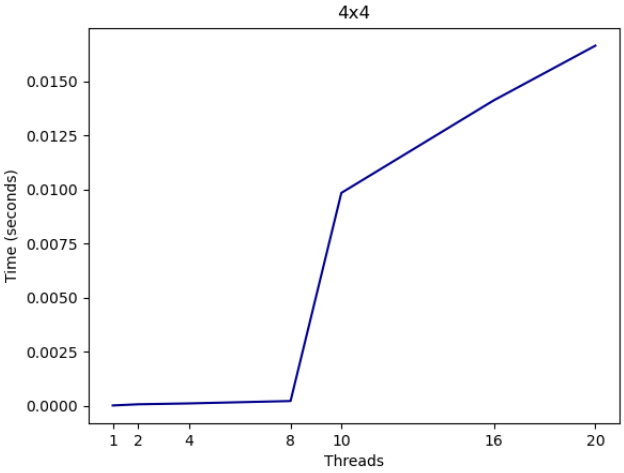
(b) *1x16 Speed-Up Graph*



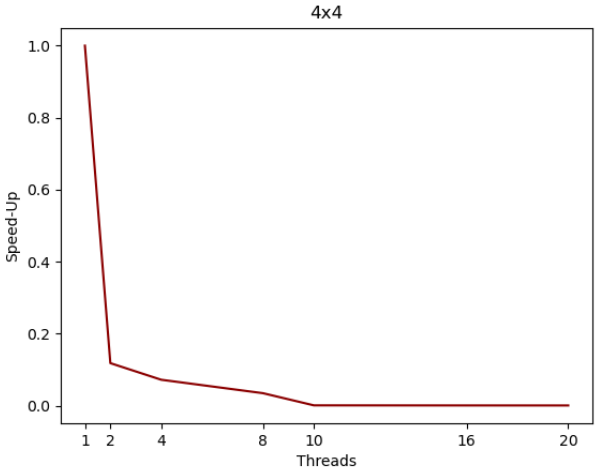
(c) *2x8 Efficiency Graph*



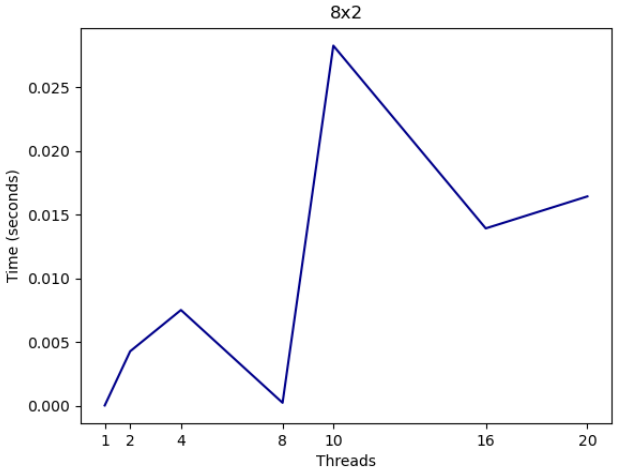
(d) *2x8 Speed-Up Graph*



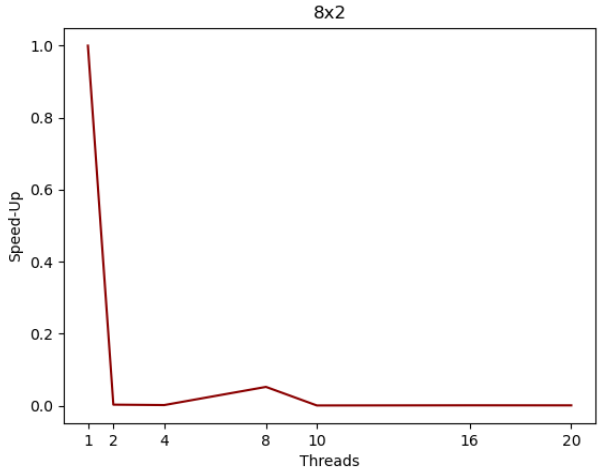
(e) *4x4 Efficiency Graph*



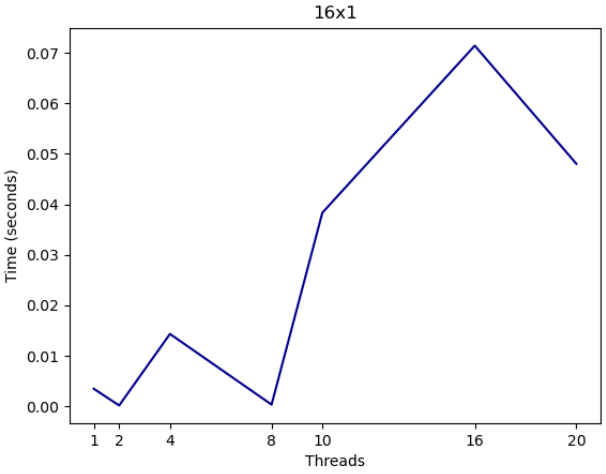
(f) *4x4 Speed-Up Graph*



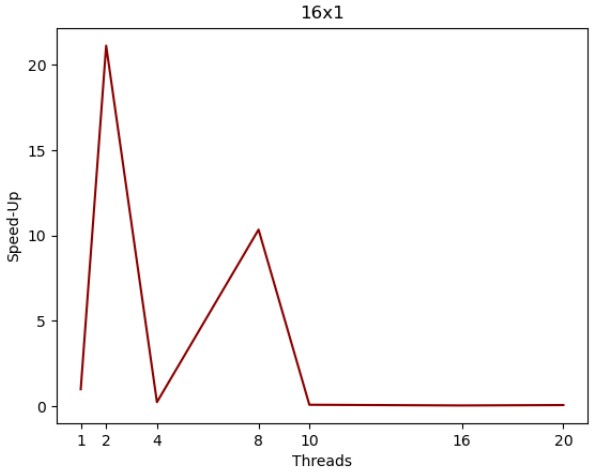
(a) *8x2 Efficiency Graph*



(b) *8x2 Speed-Up Graph*



(c) *16x1 Efficiency Graph*



(d) *16x1 Speed-Up Graph*

Figure 4: *The speed-up and efficiency graphs for each of the five additional matrix sizes. Each thread count ran for 100 iterations.*

IV. Analysis

Figure one shows the efficiency graph of the output from the 'runTimePerformance' script. There is a negative trend in the graph up until a thread count of eight is reached, from here it begins to plateau. It is clear that the parallel version of my program with increased thread count has directly affected the run time of the program. The speed up graph of the 'runTimePerformance' in figure two shows as well that eight threads had the highest speed up and had the best performance on this machine. It is unambiguous that the parallel version of this program has performed the best with eight threads on a square board.

When we then look at the smaller matrix graphs, it is strange that the efficiency has a positive trend in every matrix size. The only matrix that had a speed up greater than one for any thread count was the sixteen by one matrix for a thread count of two. This configuration saw a speed up of twenty. Each thread count was ran for one hundred iterations so it is unlikely any outliers interfered with the final results we see on the graphs.

Based on the final results, I would say that due to the overhead required to run the parallel version of Conway's Game of Life, we see little to no speed up in the smaller matrix sizes. It is unclear to me why the sixteen by one matrix performed the best. This matrix is structured similar to a one-dimensional array.

V. Conclusion

For larger matrix sizes, my parallel version of Conway's Life has a speed up of nearly eight with a thread count of eight. Due to the overhead required to run OpenMP, smaller matrix sizes see little to no speed up. If I were to run some of these performance tests again, I would use matrix sizes such as one hundred by sixteen hundred and two hundred by eight hundred, effectively increasing each of the smaller matrices by a factor of one hundred in each dimension. This would greatly increase run time, however, it would yield better results.