COMP 426
Assignment 1
Written Report

Shawn Ostad Abbasi 40191688

Gina Cody School of Engineering and Computer Science

Concordia University

The game of life is a simulation of the survival of species and their evolution. The goal of this assignment was to produce this simulation with a number of species, and strict rules; all while trying to increase the performance of the game through parallelization rather than a sequential order of execution.

For this simulation we can start off by detecting which parts would benefit from the implementation of parallelism. According to Amdahl's law, the speedup that can be obtained through parallelism is limited by the amount of work that is actually parallelizable. Because of this, recognizing what can and can't be sped up is key. Having analyzed the problem, I believe that the most parallelizable part of the simulation is the process of a cell checking its neighboring species to decide its later fate in the next generation. This problem comes across as parallelizable due to its data parallelism traits such as, performing the same computation on different data and lack of synchronization. Alongside these data parallel traits we can also argue that this problem is not only large scale but requires a lot of overhead in a sequential order of execution.

The structure of the Game of Life program is as follows: the display of the game is represented as two arrays with one acting as the current display and the other as a buffer. These arrays consist of values between -1 and 9 with -1 being a dead cell and all other numbers representing a live cell with a color. Using the currently displayed array of species, multiple threads are created with the goal of performing the rule checking on a section of the array, deciding the fate of each cell in the threads dedicated region and later placing the result/future of the cell in the buffer. In this case, making changes to the buffer and not the current array avoids race conditions and the need for synchronization thus improving speed and accuracy. After having performed computations on the displayed array of species the program then performs a number to RGB value conversion on the buffered array, stores the RGB values in a new array,

and displays the conversion through OpenGL before swapping the two original buffers before repeating the whole process again.

Execution of the program is accomplished through the control thread where in this case is the main thread. The control thread manages the execution of the program, the initialization of necessary variables such as the two arrays, usage of OpenGL functions, along with the creation and scheduling of computation threads using a fork-join paradigm. It is important to note that the control thread does not perform any of the rule checking on the cells of the arrays nor the conversion of numbers to RGB values due to these tasks being "embarrassingly parallelizable" and easily accomplishable by the usage of computation threads. Having followed the fork-join paradigm the control thread creates computation threads when necessary and may wait for the completion of the threads before continuing past some point.

Computation threads are created for the sole purpose of accomplishing an instruction of high computation on large amounts of data. In this project computation threads are accomplished through C++'s thread library. Computation threads are used for two purposes, the first being to go over all the cells in the species array and to decide the fate of the cell according to some given rules. The second usage of computation threads comes from the need to convert our buffer of numbers ranging from -1 to 9 into RGB values which again follows the idea of one instruction on different data.

While accuracy is the most important aspect of a program, performance is also necessary. In order to utilize the benefits of multithreading a decision was made to use eight threads and static load balancing. While the assignment focuses on multithreading and not multicore programming the decision to use eight threads (one per core) was still used. This value was chosen as a middle ground as to not have too little work or too much overhead. Having tested

with ten threads compared to eight caused a drop in frames per second (FPS) by nearly ten due to the overhead related to the creation and context switching of threads. For the use of the thread management the static load balancing approach was taken due to the high amount of data parallelism and common work load amongst threads. While implementing a dynamic load balancing approach may result in a slight performance boost it would also bring additional complexities that may not bring as much gain in this scenario.

To measure performance a frames per second (FPS) counter was used in order to see approximately how many generations can be computed and displayed every second. Originally before implementing optimization measures, the game ran at approximately 6-10 FPS. To correct the low FPS a couple of changes were made to the program.

The first change and perhaps the smallest was to move frequently used variables that remain unchanged to the global space rather than redeclaring the variables for every thread or every function execution. Secondly, as noted before, the two dimensional arrays that represent the species contain only numbers from -1 to 9. Knowing this, a change of the array types was made from “int” to “int8_t”. Having made this change not only can we still represent values from -1 to 9 but the size of a in8_t is only one byte compared to the four bytes needed for an int. Therefore with a 1024x768 dimensional array the size of a single buffer changed from ~3.1MB to ~786KB. The intention of this change was to lower the bytes needed for storage on the cache and inherently improve cache locality which in turn would improve run time speed. The third optimization was to reduce runtime allocation amongst threads as much as possible. Originally, each thread would create and assign values to a hashmap and vector for the sake of counting and choosing a species for dead cells to respawn as on each function call. This approach was changed for a more static one that only uses an array of a size set at compilation. Having made all these

changes the FPS had changed from 6-10 to 55-60 which demonstrates great improvement. While all of these measures were expected to increase performance there was one additional change that was surprising. As mentioned earlier the arrays representing the species contain values from -1 to 9 with each number corresponding to a color. After performing operations on the arrays the numerical values then have to be converted to RGB values and stored in a new array. The thought of adding data parallelism to this task arose due to its inherently single instruction multiple data (SIMD)-friendly structure. While the number to RGB conversion is simple I decided to test it with and without data parallelism and found that there was an increase of nearly 10 FPS with the use of data parallelism.