SERIAL IMPLEMENTATION OF THE GAME OF LIFE

Homework 1

By

[redacted]

CS 732 Parallel Computing

January, 22 2017

1 Problem Specification

Conway's Game of Life is a cellular automaton simulation algorithm that can be used to explore evolutionary concepts in many fields including biology, physics, philosophy and computer science. It was developed by the mathematician John Conway in 1970 to explore self-reproducible systems proposed by John von Neumann, the namesake for today's dominate computer architecture.

The game follows a set of basic rules which control the evolution of living cells on the game board. The game board is updated in the next generation depending on the state of the cell and its neighbors in the current generation. If a cell is alive in the current generation it survives to the next generation if it has two or three neighbors, otherwise it dies of loneliness or overcrowding. If a cell is dead in the current generation, it can spontaneously generate a living cell in the next generation if it has exactly three neighbors in the current generation.

With these simple rules the game of life can be used to explore a rich variety of physical systems. Game boards can generate patterns that behave in ways that suggest "intelligent" visual patterns. This opens the door to exploring the nature of patterns and their interpretation and the potential that biological patterns are themselves expressions of simple, fundamental rules and that any meaning is imparted by the observer and not inherent to the pattern itself.

In this assignment, the game of life fulfills a much more humble function. It serves to illustrate computation with two dimensional arrays on von Neumann architecture computers. It provides a framework for varying the size and run time of simulations in order to explore performance characteristics of computer processors. The solution described in this report accepts two arguments; the size of a square matrix that defines the game board and the maximum number of iterations the game is to run. If the game board stabilizes between generations or reaches the maximum iteration the game ends.

2 Program Design

The implementation is based largely on the provided sample code *hw1b.c*. It leverages the sample code to provide supporting logic for array creation, initialization, inspection, and performance timing. The rules for the game of life are treated as a logic kernel inserted by function into this operating framework, replacing the original matrix multiplication function.

The program creates two game boards of the given size: the ancestor and descendant. At startup the ancestor board is initialized with an initial configuration based on the likelihood that a cell is alive. A value of one indicates a living cell and a value of zero indicates a dead cell. There is a one in two chance that the initial cell is alive. After initialization, the game of life is run on the game board following the rules above and updating the descendant board with the configuration of the next generation. Through pointer updates, the descendant is promoted to the ancestor for the next round and the ancestor takes on the role of the descendant. This continues until the maximum iteration count

is reached. The program completes when the defined maximum iteration is reached and prints the number of seconds measured since the start of iteration.

The game board is initialized with a boundary of ghost cells to simplify neighbor calculation. The game of life kernel iterates through each cell on the game board and sums the value of all neighbors including the current cell. This allows the use of simple offset iteration of the three-by-three neighborhood of the cell. The rules of life are applied to the resulting sum, taking into consideration the alive or dead state of the current cell.

There are two alternative compile options for the game of life. A debug option prints each generation of game board and the neighbor counts. This provides an easy method for confirming program correctness. A "legal" option activates a logic test that causes the game to exit prior to reaching the maximum iteration if there is no change in state between the generations. Because the focus of this assignment is observation of execution performance, however, the default compilation avoids this legal state. Since the implementation inspects each cell and computes the neighbor count irrespective of the game board state, the dominate $O(n^2)$ run time characteristics do not change even if the board has stabilized. All performance numbers were produced with this as the default game play option. A Makefile allows selection of all these modes at compile time with *make*, *make debug*, and *make legal*.

3 Testing Plan

The game of life logic was validated by running small world sizes using the debug build option. Three and four iteration runs of three-by-three game boards were run in debug mode and the results of the neighborhood count, ancestor state, and descendant state where manually verified. Initial runs revealed necessary adjustments to the sub-span index for the neighbor count. Because the game board is surrounded by ghost cells, the neighbor count iterator can simply subtract one from the current cells row and column and then sum up the nine cells in the three-by-three neighborhood. Initial tests revealed errors in selecting the upper bound for these iterators. They were corrected, resulting in successful game play.

After correct game logic was confirmed, initial runs of the game for small boards were performed on a personal laptop on a 4-core Intel i7-3537U 2.0 GHz CPU and 8GB of RAM using GCC 5.4.0. The intention of the initial runs was to assess scaling properties of the algorithm. Runs of a 1000x1000 cell world at 100 (5.2s) and 1000 (47s) iterations and 5000x5000 cell world at 1000 (1171s) and 5000 (6107s) iterations were performed and demonstrated predictable scaling for each world size in response to iteration growth. On the 1000x1000 world, increasing the iteration count by a factor of ten increased the run time by a factor of ten. This scaled forward to the 5000x5000 world, a 25-fold increase in cells, resulting in a near perfect 25-fold increase in run time over the 1000x1000 at 1000 iteration test. Increasing the iteration count 5-fold saw the run time increase nearly 5-fold as well.

Test Case #	Problem Size	Max. Generations	Time Taken (seconds)
1	1000x1000	1000	8.365735
2	5000x5000	1000	220.390043
3	5000x5000	5000	1078.138904
4	10000x10000	1000	927.229598
5	10000x10000	10000	7984.956542

Table 1: Performance Results

With predictable growth established, performance improvement focused on decreasing the run time by constant factors. The first step was to activate the -O compiler flag for automatic optimizations. This resulted in a forty percent improvement in run times across the board.

The next step was to create a more compact game board to allow more efficient use of the CPU cache by using more bytes in the cache for game play. The initial implementation had simply used the double data type used by the sample program that served as the codes foundation. The double data type is an 8-byte type. Since cells are binary we can use the smallest available scalar type to produce more compact game boards and ensure each byte in the cache is used for game play. A second version of the game of life was implemented using a signed char, *gol-char.c*. Rerunning the initial tests showed a sixty percent improvement over the already optimized values. The signed char implementation formed the basis of the production tests.

4 Test Cases

The official tests of *gol-char.c* were run on a 24-core Intel E5-2680v3 2.5GHz CPU with 128GB of RAM using GCC 4.5.3 provided by UAB IT Research Computing. The prescribed tests were scheduled with a batch job designed to accept parameters for changing the world size and maximum iterations. Three jobs requesting 1-core each were submitted for each test to gather the performance averages reported in Table 1.

5 Analysis and Conclusion

The reported results demonstrate predictable execution growth within each world size across the iteration increases for this $O(n^2)$ implementation of Conway's Game of Life. Test Case #1 sets the foundation. Increasing the game board 25-fold in Test Case #2 results in a corresponding 25-fold increase in runtime. From there, extending the iteration count 5-fold in Test Case #3 increases the runtime by 5-fold. Comparing Test Case #4 and #5 shows a marginally better than 10-fold improvement for a 10-fold iteration increase, however, given more runs this would likely trend closer to the expected 10-fold runtime increase.

It is interesting to compare Test Case #2 and #4. In Test Case #2 we have a game board with 25-million cells that are traversed 1000 times. In Test Case #4 the size of the game board is increased 4-fold. Given the scaling characteristics of this implementation we would expect to see an 4-fold increase in work, resulting in a 4-fold increase in execution time. This expectation holds, $220sec \times 4 = 880sec$ is very close to the observed 927seconds.

The $O(n^2)$ performance of this algorithm is dictated by the need to inspect each cell on the game board. The only controls we have on performance are constant scaling factors derived from improvements to the array representation, optimizations in the loop structure, or reduction in the number of operations executed in each loop. As the discussion in Section 3 demonstrated, the compiler optimization and array representation each provided constant factor improvements in run time. The nature of this algorithm suggest it will respond well to other performance enhancements derived from problem devision. While we can't overcome the growth of execution time in response to game board increases, we can optimize how the work is distributed in order to limit the execution time.

The clean scaling of this algorithm likely results from the fact that it follows optimal cache utilization patterns. Each cell is read sequentially row by row, ensuring that the memory load characteristics of the cache can benefit the code. As additional cells are read, they will have already been preloaded into the CPU caches. Also because references to other areas of the game board are limited to nearby cells in the prior or subsequent rows, our problem sizes ensure this distance is never more than 10KB away. Due to the size of the cache and the least-recent replacement policy, these adjoining cells are very likely to still be resident in the cache. Once we move past a row, we will only use it once more in the processing of the next row. After that it can be ejected from the cache for a given generation. In other words, even though the game boards are up to 100MB in size, the working set is no more than 3x10KB for the ancestor and 1x10KB for the descendant. This ensures ample room in the cache for efficient data access.

The code was implemented using a common coding paradigm of building a framework of library routines and then simply replacing the kernel of the logic of the system to limit the amount of software needed and potential for errors. Changing a matrix multiply program to a game of life program was accomplished almost entirely by replacing the matrix multiply function with a game of life function. Alternating between different data type representations for the world maps could be improved by providing compiler preprocessing variables that could change the data type of variables without having to change the core logic of the program. This would allow the program to change from an implementation that uses a *double* to one that uses a *signed char* with out disturbing the structure of the application, unlike the current approach that duplicates the code in a new program file.

The change from a double representation to a signed char was very simple and eliminated 7 bytes per cell of unused space. This allowed larger world maps to reside in cache. Further representation improvements were considered to house both generations in the same byte. This could have been accomplished

by using bit masks to split the byte into a high and low bit field. This would allow both the ancestor and descendant to occupy the same space in memory and improve update efficiency. However, this would complicate the code and demand much more significant implementation changes to the logic for computing neighbor counts. Similarly, vector representations were considered that could leverage the large AVX registers in modern processors. However, this would require counting neighbors in surrounding bits. While possible, the implementation complexity suggest these are best reserved for when world sizes get so large only the most efficient representation makes their execution tenable.

Further reading suggests more memory efficient implementations are possible by avoiding the need to maintain a full game board for the descendant. Simply using buffer lines as temporary storage allows the descendant results to be stored in the ancestor once processing on the current row is complete. Additionally, vector implementations of the game board are possible by recording only the living cells allowing for compact representation of very large game boards.

This assignment introduced Conway's Game of Life as a computational device for exploring execution growth as a function of problem size. Efficiency improvements to the implementation were introduce with compiler optimizations to enhance instruction sequencing. Additionally, creating more compact representations of the game board optimized use of memory. The patterns of execution, stepping through the game board sequentially and the limited dependencies on other areas of the game board, suggest there is ample room for improvement by dividing the problem into smaller chunks.

6 References

1. Mathematical Games, Martin Gardner, http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt access Jan 23, 2017
2. Conway's Game of Life, Wikipedia, on-line, accessed Jan 22, 2017
3. A Simple Makefile Tutorial, Maxwell, http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/, accessed Jan 16, 2017
4. Extending builds for debug and production, http://stackoverflow.com/a/1080180, accessed Jan 16, 2017
5. Code samples hw1a.c and hw1b.c , Puri Bangalore, Jan 16, 2017