# USING A LINUX AND BATCH SYSTEM: THE GAME OF LIFE

Homework 2

By

Osho Robin

CS 732 Parallel Computing

February 6, 2017

# 1 Problem Specification

Conway's Game of Life remains the illustrating device for computational analysis in this assignment. The solutions described in this report expand on the initial computational runs of the Homework 1, adding the compute resources of the Alabama Supercomputer Center (ASC) and the performance impact of compiler choices. Additionally, the analysis includes commentary and reflections on exponential growth inspired by a recent Communications article on this topic[1].

# 2 Program Design

The implementation remains unchanged from Homework 1. The game board is initialized and allowed to run through its $O(n^2)$ algorithm, inspecting each cell of the $n \times n$ game board and evolving the game over the given iteration count. This assignment is focused on the impact of compiler select, compiler options, and platform selection on the runtime of the application.

Additional supporting code in the form of a a submit script and batch job script were developed to simplify submission of the required analysis runs. The submit script queues three jobs for each execution scenario, by default game boards of $1000 \times 1000$, $5000 \times 5000$, and $10000 \times 10000$ for the prescribed run lengths. The submit script also calculates appropriate memory requirements based on the use of two 1-byte-per-cell game boards.

Compile variations using different compiler choices and compile time flags were accomplished using the CC and CFLAGS variable overrides to the Makefile developed in the first assignment. Choosing an alternate complier simply required running make CC=icc gol-char or adding additional optimization via make CFLAGS=-O3 gol-char.[2]

# 3 Testing Plan

With linear scaling of run time for a given problem size established in Homework 1, performance improvement focused on decreasing the run time by changing the compiler and compiler options. The GCC 4.8.5 and Intel 15.0.3 compilers were compared with the standard -O optimization flag. The -O3 compiler option was introduced for GCC compiles in order to build competitive executables between GCC and Intel compliers based on performance tuning recommendations from a NERSC compiler comparison study.[4] Jobs using these options were submitted on ASC's DMC cluster and UAB's Cheaha cluster. Code was synchronized across platforms using Git and the GitLab server provided by the UAB CS department.

| Test Case # | Problem Size | Max. Generations | Time Taken (seconds) | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | Cheaha | | | DMC | | |
| | | | gcc -O | gcc -O3 | icc -O | gcc -O | gcc -O3 | icc -O |
| 1 | 1000x1000 | 1000 | 8.366 | 4.452 | 2.206 | 16.663 | 8.632 | 4.379 |
| 2 | 5000x5000 | 1000 | 220.390 | 112.788 | 57.585 | 402.932 | 214.347 | 109.009 |
| 3 | 5000x5000 | 5000 | 1078.139 | 496.756 | 286.971 | 1815.38 | 1002.49 | 543.973 |
| 4 | 10000x10000 | 1000 | 927.230 | 443.181 | 217.159 | 1656.02 | 856.628 | 435.779 |
| 5 | 10000x10000 | 10000 | 7984.957 | 3682.82 | 2088.24 | | 7916.35 | 4355.02 |

Table 1: Performance Results

# 4 Test Cases

The official tests of gol-char.c were run on a 10-core Intel(R) Xeon(R) CPU E52670 v2 @ 2.50GHz CPU with 128GB of RAM using GCC 4.8.5 and Intel 15.0.3 (GCC 4.8.5 compat) provided by ASC and a 12-core Intel E5-2680v3 2.5GHz CPU with 128GB of RAM using GCC 4.8.5 and Intel 15.0.3 (GCC 4.9.3 compat) 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 continue to demonstrate linear execution growth within each world size across the iteration increases for this $O(n^2)$ implementation of Conway's Game of Life. The most significant insight from these comparisons is the speed advantages gained from the compiler and the choice of CPU. The test results show that using the

Intel 15.0.3 compiler with standard -O optimization provides a 1.8x speed-up over the GCC 4.8.5 with even the more aggressive -O3 optimization enabled. GCC 4.8.5 is the system provided GCC compiler on CentOS 7. Both compilers were released during the 2015 calendar year.

The CPU choice also has significant impact on performance. This choice provides an additional 2.5x speed-up from the newer generation of chips on Cheaha (2014Q3) versus the DMC (2013Q3). While the nominal clock speeds of both chips are 2.50GHz, there is 12% increase in memory bandwidth and a 16% increase in cache size in the later model CPU on Cheaha. These differences alone likely cannot fully account for the 2.5x speed up between the platforms. A potential difference that could account for such a significant difference is the speed of the memory chips themselves. The DMC CPU supports DDR3 RAM ranging from 800 to 1800 mega transfers per second. The Cheaha CPU support DDR4 RAM ranging from 1600 to 2100 mega transfers per second. If the each system is populated with speeds from the lower end of their range, a common cost saving approach, then the systems would have a factor of two difference in their memory performance. Further investigation of installed memory and the impact of its data transfer rate on the game of life is warranted.

Taken together, the impact of compiler and CPU made the difference between failed execution on the DMC of the largest system, $10000 \times 10000$ and
10000 iterations, using standard GCC optimization, due to the runs exceeding their 4 hours alloted time. This compares to the successful execution of the same model on Cheaha completing in 35 minutes using standard optimization and the Intel compiler.

The disparity between the highly optimized GCC compiles and standard optimization Intel compiles was investigated with a cursory inspection of the the assembler code generated by each compiler for the evolveWorld() function where the bulk of the computation time is consumed. Both compilers generated their assembler code using CFLAGS="-Wa,-adhln -g" in addition to the -O and -O3 optimization flags for Intel and GCC, respectively. Comparing the GCC and Intel assembler output revealed better loop unrolling by the Intel compiler in the the code section that counts the neighbors of the current cell. Given that this code is executed nine times for each cell, this unrolling could be significant. In general the Intel compiler assembler seemed to be arranged to reduce compare and jumps. There was significant use of MMX vectorization during the array initialization but given this happens only once in the code its impact on performance is expected to be minimal. A further study of the generated assembler could provide greater insight. It would also be worth comparing the performance of later versions of both compilers. A recent report from MIT on optimizations using the LLVM compiler also suggests potential improvements may be gained using this compiler.[3]

## 5.1 Special Focus: Exponential Growth

"Exponential Laws of Computing Growth" by Peter J. Denning and Ted G. Lewis published in the January 2017 issue of Communications of ACM discussed several characteristics of exponential growth that have sustained the phenomenal growth of the

computing industry for the past fifty years.[1] Moore's Law refers to the component doubling rate on computer chips through improvements in manufacturing. (Ans 1) This law was derived from empirical evidence in the in the early 1960's and used to successfully predict the expected component density of chips in 1975. Since then this exponential growth law has held and predicted a doubling in component density every 18 to 24 months. This doubling at first lead to continued clock rate increases but has now transitioned to additional cache and computational hardware located on the CPU, driving today's growth in multi-core technology.

Dennard scaling refers to the the power density of circuits remaining the same as component sizes shrink. (Ans 2) The shrinking components shorten the distance that electrical currents must travel in circuits. This scaling effect enabled significant increases in the speed of processors because smaller circuits could support shorter clock cycles. Dennard scaling served as the first significant technology for maintaining exponential growth. Since about 2000, the compute densities have increase to the point where this scaling is no longer effective due to the heat generated.

The S-curve model is an economic model that describes the adoption of new technologies. (Ans 3) Adoption initial grows exponentially and then reaches an inflection point where adoption levels off. This reflects transitions through innovator, early adopter, late adopter, and laggard phases of adoption. As new technologies arise they introduce their own new S-curve. It is therefore possible to jump from one S-curve to the next and maintain an exponential growth path.

Moore's Law is not dead. It continues to apply to the exponential growth in compute density and is maintained by continued jumps to new technologies as the S-curve on current manufacturing technologies levels out. (Ans 4)

The authors identify three levels of technology at the chip, system, and community level that are needed to sustain exponential growth. (Ans 5) As chip performance grows exponentially, it's attendant support systems must grow exponentially or the CPU will be left without work. As the chip and system performance grow, the community of users must grow exponentially to absorb the cost of development and production. Without exponential growth in all three areas, one would become a bottle neck and stifle growth in the other domains. The authors argue that the conditions which sustain exponential growth at all three of these levels exist to sustain exponential growth well into the future.

# References


[1] Peter J. Denning and Ted G. Lewis. Exponential laws of computing growth. Commun. ACM, 60(1):54–65, December 2016.

[2] Free Software Foundation. Gnu make manual, 2016.

[3] Larry Hardesty. Optimizing code, 2017.

[4] NERSC. Compiler comparisons, 2016.