Parallelizing Conway's Game of Life

Samuel Leeman-Munk & Tiago Damasceno

November 4, 2010

Abstract

A brief explanation of Life and its parallelization. Lab included.

Contents

1 Background

Game of Life is a cellular automaton formulated by the British mathematician, John Orton Conway in 1970. This “game” does not require any players. The user just sets up the initial configuration and leaves the game to evolve on its own. Conway invented Life as a simplification of a much more complex model by John Von Neumann. Von Neumann had invented his automaton in a search for a hypothetical machine that could build copies of itself, and Conway’s Life can do the same. What particularly interests the computational community is that Life is Turing Complete, meaning it can compute any algorithm that a conventional computer can. Life and other cellular automata are famous for their “emergent behavior,” The process by which elaborate patterns and behaviors can emerge from very simple rules.

1.1 Examples of patterns

Many patterns have been discovered over the years. Still lives are simple static patterns that do not change, such as blocks, beehives, loaves and boats. Oscillators are a superset of still lives. They cycle in between shapes and never die. These include blinkers, toads, beacons and pulsars. A particular type of oscillator is the Spaceship. A spaceship pattern is so named because as it oscillates it translates itself across space (the board). One of the most well-known life patterns, the glider, is a famous spaceship. There are many other pattern types, such as Methuselahs, Guns, Puffers and Rakes.

1.2 Rules

Conway’s game of Life revolves around 4 simple rules:

1. Any living cell with less than 2 live neighbors dies, as if it was caused by under-population.
2. Any living cell with more than 3 live neighbors dies, as of it was caused by over-population (overcrowding).
3. Any living cell with between 2 and 3 neighbors lives on to the next generation.
4. Any dead cell with 3 living neighbors will be "born," or become alive.

2 Implementation

Life the computer program consists of a small loop that repeatedly operates on a two-dimensional array. For each step, a loop goes through each column, and decides the new state of each cell in the column one at a time. Then all the cell changes are applied at once, and the loop starts again.

Note that in deciding the new cell state, the program does not immediately change the cell to the new state. If it did, the cell processed next would change based on the new state rather than the original state. This would not accurately reflect Conway's game of Life, which assumes all the cells change state simultaneously. The new values of the cell states are stored in another two-dimensional array and copied[1] over to the original array at the end of each iteration.

The theoretical Game of Life takes place on a board of infinite size. In order to best approximate an infinite game board, Life wraps at the edges. To allow the cells to interact with each other across the edges of the board, Life pads the main array with an extra layer of invisible cells. Between iterations, these cells are simply updated with their corresponding cells on the other side (e.g. the invisible cells past the leftmost border map to the visible cells at the rightmost border).

Each cell in Conway's game of life needs to know the states of its eight neighbors and itself. That's nine cells, no matter how many cells the whole board contains. Thus,

[1]Do they have to be copied? Those advanced programmers among you, what might be a more efficient way to handle this?

the required computations are directly proportional to the number of cells, i.e. Life scales linearly (O(n)).

3 Parallelization

Life is an instance of the structured grid Dwarf of Berkeley's thirteen dwarfs [1]. Weather simulations are another example of a structured grid dwarf. In particular, Life is a 2D grid in which each cell is concerned only with its direct neighbors, a simple instance of its dwarf. Distribution of labor in a structured grid Dwarf like Conway's Game of Life is simply a matter of splitting the board's cells among the threads. Of course, the most intuitive way for a human to split a structured grid into pieces is to break it into chunks of approximately equal size and of the same shape as the original grid. More intuitive to program, however is to slice the grid into rectangles, giving each thread a set of rows or columns to evaluate (Why do you think this is?).

Once the board is divided, work continues much in the same way as serial. In shared memory, border cases are trivial, and in distributed memory either each slice receives an "invisible" border of the cells that border it under the control of other processes, or each process receives its own complete copy of the current array at the beginning of each iteration2. Like any structured grid Dwarf, Life is tightly coupled. Because each iteration requires a rebroadcast of the current board, a process that gets to the end of an iteration first must wait for all the other processes to complete that iteration before it can continue. If the entire board is broadcast to each process, communication required for Life is proportional to both the number of cells (the problem size) and the number of processes, making Life relatively communication heavy. Being an extremely simple model with no potential for roundoff error, Life sacrifices no precision in its parallelization.

4 Files

Makefile Contain instructions to compile the Game of Life automatically, connecting all the necessary files, flag and etc.

Life.c Initializes Life object which contains all the variables used in this program. This file also has a loop that runs through N-Generations. The loop contains all the instructions (function calls) to run one generation of the simulation.

Life.h Maintains all the relevant functions used by Life.c

Defaults.h This file contains the definition of all the variables, used in the program:

XLife.h Contains all the necessary functions to create a X11 window that display the right number of rows and columns based on the information that is set on the other Life files. "Dot in" files (*.in) This type of file contains X and Y coordinates for all the alive cells and also the dimension of the world. The first two numbers in the first line is the world's X and Y dimensions in which the simulation of Life will occur. The other pair of numbers on the lines below are the X and Y coordinates for all the alive cells. These files can be used as inputs for the Game of Life.

5 Compiling & Running

Before running the program you must build it. Make sure to be in the relevant folder, and type the command "make Life", in the terminal window. This command will create an executable file named Life. To run the program use "./Life [Rows] [Columns] [Generations] [Display]" or use "./Life" combined with the Command Line Options described below. For information on automatically generating scaling and parallelization graphs for Life, please refer to the UPEP module Multidimensional Benchmarking with PetaKit.

6 Command Line Options

```
-h|--help             The help page.

-c|--columns number   Number of columns in grid. Default: 105

-r|--rows number       Number of rows in grid. Default: 105

-g|--gens number       Number of generations to run. Default: 1000

-i|--input filename Input file. See README for format. Default: none.

-o|--output filename  Output file. Default: none.

-t[N]|--throttle[=N]  Throttle display to Ngenerations/second.Default:100

-x|--display          Use a graphical display.

--no-display          Do not use a graphical display.
```

7 Output

When utilizing X11 the output should resemble the picture on the right. There are examples of patterns in the program's folder, such as the Oscillators, Gosper Glider Gun and etc. To visualize the examples use the "-i" option (input file). If the output file option is used, a file will be created with the X and Y coordinates of the cells that are alive at the last generation simulated.

8 Laboratory

8.1 Prerequisites

- Prior experience using UNIX command-line interface.
- Knowledge of how to use an accessible parallel processing environment[2]

8.2 Materials

- Computer (1 per student)
- Parallel Processing Environment[3]
- UPEP Life Module

8.3 Procedure

1. From *Multidimensional Benchmarking with PetaKit* take the StatKit zip file[4] and unpack it into the directory where the folders with your source code are.
2. In each source code directory, run the following command:

   ```
   make
   ```

   This will build your executables.
3. If you are running on a cluster that requires a scheduler, construct a template for submitting jobs to the scheduler[5]. Making a script template is as simple as filling in all the directives you'll want to be passed to your scheduler, for the values that will change substituting variables specified in the StatKit documentation (perl

[2]In other words, do not attempt this lab unless you or someone with whom you are working can access a parallel machine that he or she can use. This includes but is not limited to clusters and multicore personal computers. Be aware that cluster computers are highly complicated devices, and most of the commands coming up will *not work* if you just copy them directly into your command line. In short, **Read the instructions carefully.**

[3]A cluster is ideal for this purpose, but if one is not available, nearly all modern computers have at least two cores, which can serve as a very small-scale parallelization environment

[4]https://shodor.org/media/content//petascale/materials/UPModules/pkit/StatKit.zip

[5]If you don't know how to submit jobs to your scheduler, speak with someone who does

stat.pl –help). Make sure to include in your script where appropriate $special $prep and $main, in that order and each with its own line. These form the main command that actually runs your program. If you still don't understand, try looking at the existing templates in the script_templates directory to get a better idea of the format, and remember the StatKit documentation.

4. Save your template in the StatKit folder in the script_templates directory. Remember what you named your template. We will refer to it as "myTemplate.sh". We'll be referring to your cluster's scheduler as myScheduler, also.
5. Now you're ready to begin making performance scaling graphs!

Serial:

Scale the serial code, running over several problem sizes and collecting the times.

In the StatKit folder(stats), run the following command:

```
 perl stat.pl --cl '../GoL_Serial/Life -r 10 -c $problem_size -t 1000 --no-display' \
-t <myTemplate.sh> --problem_size 100,1000,10000,50000,100000 \
--repetitions 5 --proxy-output --tag mySerialLife \
--scheduler <myScheduler>
```

Remember to replace values in angle brackets (<>) with values appropriate to your cluster!

problem_size is the most significant variable in a program. The problem size is, in general, the variable that has the greatest effect on runtime. For instance, in Life the problem size is the number of cells. Strictly speaking, this command to StatKit presents the problem size as a tenth of what it actually is[6]. The current version of StatKit doesn't support arbitrary functions in templates, so this is a programmatic convenience that will not affect results.

Repetitions decides how many times you'll run at each problem size. Proxy output simulates the PetaKit output so the user doesn't have to program it into his or her program to be benchmarked.

[6]why?

While you wait for these runs to complete, consider what sort of scaling you will expect. Scaling is how the runtime of a problem changes as one increases some part of it. In this case, you're scaling over problem size.

Once all the runs are finished, return to your main directory and run:

```
 perl PlotKit.pl \
--independent problem_size  --dependent walltime \
--datafile stats/output.txt -o mygraph.ps mySerialLife
```

StatKit outputs its results into a file called output.txt. Output.txt remembers the variable names so you don't have to remember which column they're in. Take a look at output.txt to see how it's layed out. PlotKit.pl collects the relevant data, averages repeats and builds a graph. Independent and dependent should be self-explanatory, they decide which variables populate the x and y axes respectively. The last option must always be the tag(s) of the runs you want to see.

In whatever way you can on your machine, look at mygraph.ps[7]. If you can view graphical output directly on your machine, feel free to omit the '-o mygraph.ps" option to get a graph sent directly to your screen.

Shared Memory Parallelism:

Now that we have some parallelization, we'll scale Life varying the number of that can be used on one machine. Because multiple cores on a single processor share the same memory, this is known as *shared memory parallelism.*

Be aware that if you are not running through a scheduler you will need to define OMP_NUM_THREADS to externally restrict the number of threads a shared memory parallel program uses. The "–special" argument will let you do this. In Bourne Again SHell (BASH), the value you'll want to give it is as follows:

```
 --special 'OMP\_NUM\_THREADS=$threads'
```

In the StatKit folder(stats), run the following command:

[7]If your computer can't handle .ps, try ps2pdf, or for Windows GhostScript http://pages.cs.wisc.edu/ ghost

```
 perl stat.pl --cl '../GoL_OpenMP/Life -r 10 -c $problem_size -t 1000 --no-display' \
-t <myTemplate.sh> --problem_size 100,1000,10000,50000,100000 --threads 1-4\
--scheduler <myScheduler> --repetitions 5 --proxy-output --tag myOmpLife
```

```
 perl PlotKit.pl \
--independent threads  --dependent walltime --split problem_size \
-o mygraph.ps --datafile stats/output.txt myOmpLife
```

Notice the new "split" option, which allows us to "split" a graph into a different graph for each value of a variable. Observe your graph. How does the problem scale over the number of threads? How does the scaling change with respect to problem size? What does this tell you about parallel processing?

Distributed Memory Parallelism:

Now we'll compare distributed and shared parallelism. With pure distributed parallelism each thread gets its own processor.

Distributed Memory implementation is more challenging because it depends more heavily on cluster implementation. The following command is just an example, and you may need to work with your cluster manager to figure out the correct command-line template.

```
 perl stat.pl \
--cl 'mpirun --bynode -np $processes --hostfile <myHostFile> \
 ../GoL_MPI/Life -r 10 -c $problem_size -t 1000 --no-display' \
-t <myTemplate.sh> --problem_size 100000 --processes 1-8\
--scheduler myScheduler --repetitions 5 --proxy-output\
--tag myMPILife
```

For comparison, generate some more shared memory data.

```
 perl stat.pl --cl '../area-omp 0 1 $problem_size' \
-t myTemplate.sh --problem_size 100000 --threads 1-4 --scheduler myScheduler\
--repetitions 5 --proxy-output --tag myOmpLife2
```

```
 perl PlotKit.pl \
--independent threads  --dependent walltime --split tag \
-o mygraph.ps --datafile stats/output.txt myOmpLife2,myMPILife
```

To graph more than one tag, combine them with commas. In this example we then split over tag to compare the two runs. Notice that although MPI uses processes, PlotKit.pl still refers to the output as threads. For purposes of parsing convenience, StatKit always outputs the number of units working on a problem as threads.

From this graph, what do you notice about distributed memory parallelism as compared to shared memory parallelism? Why do you think this is? What advantages do we get for distributed memory parallelism?

8.4 Questions

1. In Shared memory, what problem size does Life need for a distinct shape to appear when scaling over the number of threads?
2. Does Life scale differently with different shapes of board? Would setting columns at 10 and varying the number of rows give a significantly different result? What about a square board? Test your hypotheses.
3. What shape of graph would you expect if you varied the problem size proportionally to the number of processes or threads?
   (a) Try it, run OMP or MPI stats again with –pfunc linear and –tag 'weak scaling' Don't forget to use PlotKit.
   (b) What shape did you get?
4. Does the content of the board affect the runtime of Life? Try using one of the provided input files to find out.
5. Given your scaling information, which version of Life is best in which situation? For instance, Which runs most quickly when only one core is available? What

about when several cores are available, but each core is on a different compute node? Explore, and explain your findings.

8.5 Assessing Student Understanding

A student that has or can quickly generate performance graphs for each parallel version of AUC and can answer all of the questions above has gained full understanding of this lab.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] S. Leeman-Munk, A. Weeden, A. F. Gibbon, B. Johnson-Stalhut, M. Edlefsen, G. Schuerger, D. Joiner, and C. Peck. SIGCSE Milwaukee, 2010.