Report
Cellular automaton

Version 1

The GLUT library periodically calls the callback function myTimerFunc(), which is defined in this code. It is employed to update the game's state and display the new state on the screen..

To prevent compiler warnings about unused variables, the function first ignores the value argument (with (void) value;). The timer is then reset by calling glutTimerFunc() with a delay of 100 ms and myTimerFunc()'s callback function with an argument of 0.

It then calls generationVI() to create the game's grid's subsequent generation.
The function generationVI() is in charge of creating the game's grid's subsequent generation. The first thing it does is determine whether the version is single- or multi-threaded. It calls threadFunc() with a nullptr argument if it is single-threaded. If it is multi-threaded, it makes a thread for every group of rows in the grid and calls threadFuncVI() with a ThreadInfo argument in each thread. The section of rows that the thread is in charge of updating are described in ThreadInfo.

Each thread executes the function threadFuncVI(). By calling cellNewState(), which determines the new state of the cell based on the game's rules, it updates the state of each cell in its segment of rows. The new state of the cell is simply set to newState if the game is in black and white (colorMode == 0) or the cell is dead (newState == 0). A cell's age is increased by one if it is alive and in color mode otherwise, unless it has reached the maximum age in which case it remains the same.

GenerationVI() waits for the threads to finish running by calling pthread_join() on each thread after each thread has finished updating its segments of rows. Then, after sleeping for a predetermined amount of time (measured in microseconds), it swaps the grids (so that the current grid becomes the next grid and vice versa) and increases the generation counter

To display the updated grid state on the screen, the function calls myDisplay() at the end.

Version2

Multiple threads are used to run the simulation, which speeds up computation. The simulation grid is split up into a number of separate regions, and each thread is in charge of updating a different area of the grid.

Two arrays, currentGrid and nextGrid, are used to represent the simulation grid. The former stores the grid's current state, and the latter, which is computed based on the game's rules, stores the grid's upcoming state. The next state of the cell (i,j) is determined by its neighbors using the cellNewState(i,j) function.

A set of ThreadInfo structures, one for each thread, are initialized by the main thread and passed to the threadFunc function. The start and end rows of the grid region each thread is in charge of updating, the thread's index, and a mutex lock used for thread synchronization are all included in each ThreadInfo structure.

Each thread iterates over the region of the grid that is assigned to it in the threadFunc function, computing each cell's subsequent state. The main thread switches the current and following grids and updates the generation count once all threads have finished updating their respective regions. When the current thread completes its next iteration, the main thread releases the locks on all other ThreadInfo structures save for the one that belongs to it. As a result, the other threads must wait for the current thread to complete its next iteration.

Version3

The main function begins by using calloc to initialize all of the memory to zero before allocating memory for an array of ThreadInfo structures. Each ThreadInfo structure's index member is set to the corresponding thread's index by the for loop.

With a pointer to the structure and the address of the threadFunc function as arguments, the second for loop's pthread_create function creates a new thread for each ThreadInfo structure. Random coordinates are generated, locks are obtained for the cells at those coordinates, the states of the cells are updated, and then the locks are released by the threadFunc function. Prior to repeating the loop, the usleep function pauses the thread for the specified number of microseconds.

Cells are locked and unlocked using the getLocks and freeLocks functions. Each cell has a corresponding mutex lock and is represented as a two-dimensional array of integers. The cell's row and column coordinates are used by the getLocks and freeLocks functions to lock or unlock the cell's mutex lock. The function locks the mutex locks of the cell and its eight neighbors if the cell is not at the edge of the grid. The function only locks the cell's mutex lock if the cell is on the edge of the grid.

When the simulation is running in multithreaded mode, the mutex locks are destroyed using the #if VERSION == MULTI_THREADED block.

Scripts:

- build.sh

By iterating through an array of version names and compiling the corresponding source files with the g++ compiler, my script creates multiple versions of a C++ project

It starts by defining the names of all project versions in an array called "versions." The "for" loop that follows iterates through each version in the "versions" array.

Within the loop, the script uses the "cd" command to switch to the directory of the current version and then uses the g++ compiler to create the executable for that version.

- bash.sh

My script starts a cellular automaton process and uses a named pipe to communicate with it. The width, height, and number of threads to use for the automaton grid are the first three command-line arguments that the script parses.

The cellular automaton process is then started by the script, which directs its output to the named pipe "/tmp/cellpipe" and provides it with the arguments for its width, height, and number of threads.

The function "send_cmd," which is defined in the script, sends a command to the process and waits for a response. This function reads the response from the process' standard output after writing the command to its standard input.

My script then enters a loop that uses the "send_cmd" function to send commands to the cellular automaton process after reading them from standard input. The script can be used to change the rule number, activate or deactivate colors, draw horizontal lines, and modify the simulation speed, among other commands

My script sends the "end" command to the cellular automaton process and exits the loop when it receives the "end" command, after which the loop continues until the "end" command is received.

The script deletes the named pipe to complete the cleanup.

- bash_v2.sh

My script that starts and controls numerous cellular automaton instances. WIDTH, HEIGHT, and THREADS are three arguments that the program accepts from the command line. With these arguments, the script launches the cellular automaton program and creates a named pipe for communication with it.

In order to read user input and execute commands for the chosen cellular automaton process, the script then enters a loop. Associative arrays are used by the script to keep track of the process index, PID, width, and height for each launched process. my script deletes the entry for a process when it exits from the arrays.

The following commands are among those that are supported: "end" to end the process; "faster" and "slower" to speed up or slow down the simulation; "rule [number]" to set the simulation's rule; "color on" and "color off" to enable or disable colored output; and "line" to toggle whether or not to display a line. The script outputs an error message in the event that an invalid command or index is entered

3.4.3 Extra credit 3: Discussion (up to 5 points)

The issue of synchronizing access to the entire grid with the rendering thread is comparable to the synchronization issue. Two threads, one that produces data and the other that consumes data are present in this problem. The consumer thread retrieves data from the shared buffer that the producer thread has created and processes it. It is difficult to prevent the consumer from attempting to consume data that has not yet been produced and the producer from accessing the buffer at the same time.

In the simulation of the game of life, the grid-updating threads can be thought of as producers, whereas the rendering thread can be thought of as a consumer. The updated grid is created by the threads, which are then consumed by the rendering thread and shown on the screen. As a result, we can use a shared buffer between the updating threads and the rendering thread to synchronize access to the entire grid with the rendering thread. The rendering thread can read the updated grid from the shared buffer and display it after the updating threads have written it there

Using a mutex lock to secure the shared buffer and two condition variables one for updating threads to alert the rendering thread when the buffer has been updated and another for the rendering thread to alert the updating threads when the buffer has been used we can implement this solution. The first condition variable is used by an updating thread to notify the rendering thread when the grid needs to be updated, and the second condition variable is then awaited. The rendering thread waits for the first condition variable until it has finished rendering the grid before notifying the updating threads using the second condition variable.

This strategy prevents the rendering thread from attempting to render an out-of-date grid and the updating threads from simultaneously accessing the grid