Fast Fungi: an Assessment of Runtime Scalability in a Parallelized Simulation of Mycelium Growth

COMP445 Final Project, Spring 2021

Aron Smith-Donovan

Table of Contents

page #

List of Tables

page #

List of Figures

page #

1 Introduction

1.1 Background

Fungi are multicellular, spore-producing organisms, and certain species sprout mushrooms as fruiting bodies; spores released from mushrooms are capable of asexual reproduction and will germinate into new fungal growths with sufficient nutrition. Although mushrooms are often the most visible part of the organism, the body of the fungus itself is composed of the mycelium (pl. mycelia), a massive underground network of thin, branching tubules know as hyphae (sing. hypha).

Fungi grow *apically*, with each hypha extending only at the tip without increasing in diameter. Hyphae are responsible for outward growth, allowing the fungus to seek out unexploited areas where it (or another fungal organism) has not already grown. Because of this resource-seeking pattern, mycelia typically grow in a radial pattern as hyphae branch further from the original site of the spore and die where they can no longer be sustained. Mushrooms appear where the hyphae have had time to mature and become capable of fruiting but not enough time has passed for the soil to become depleted; this is why naturally-occuring rings or arcs of mushrooms occur, also called "fairy rings".[1]

1.2 Problem description

The growth of mycelium into these fairy rings is complex and depends on many factors; nonetheless, because of its spatial component, the area can be modeled as a square, two-dimensional *structured grid*. Structured grids are a computational tool representing some physical space and composed of points with relations to their neighbors. To begin building this simulation, we create a 2D array of integers, where each cell contains a number corresponding to some status of the fungus at that location.

To describe the growth cycle over time, we create a rule set describing the conditions for a cell to change state, such as a young hypha growing into an adjacent cell or a mature hypha sprouting mushrooms. The grid is updated sequentially on each time interval until a determined end point has been reached, and the value of a given cell is dependent on the grid update steps and the value of other nearby points. The cell states over sufficient time intervals will approximately describe the fungal radial growth cycle and can be expected to recreate the ring pattern.

Rather than incorporating factors external to the growth cycle, e.g. weather or soil quality, we will design the rule set so as to create a *discrete stochastic system*. This system will define the probabilities of defined transitions occurring on a given change in time interval and use a *random number generator* (RNG) to create variations in the growth patterns; the use of this stochastic component will approximate the impact of excluded factors and ensure a very high number of simulations before a progression is duplicated, thereby making it possible to generate artificial sample data.[2]

2 Implementation

2.1 Solution description

I successfully implemented a structured grid model and stochastic rule set to simulate the mycelium growth

Table 1: Cell states

0	EMPTY	empty ground containing no spore or hyphae
1	SPORE	contains at least one spore
2	YOUNG	young hyphae that cannot form mushrooms yet
3	MATURING	maturing hyphae that cannot form mushrooms yet
4	MUSHROOM	older hyphae with mushrooms
5	OLDER	older hyphae with no mushrooms
6	DECAYING	decaying hyphae with exhausted nutrients
7	DEAD	newly dead hyphae with exhausted nutrients
8	DEADER	hyphae that have been dead for a while
9	DEPLETED	area whose nutrients have previously been depleted by growth
10	INERT	inert area where plants cannot grow

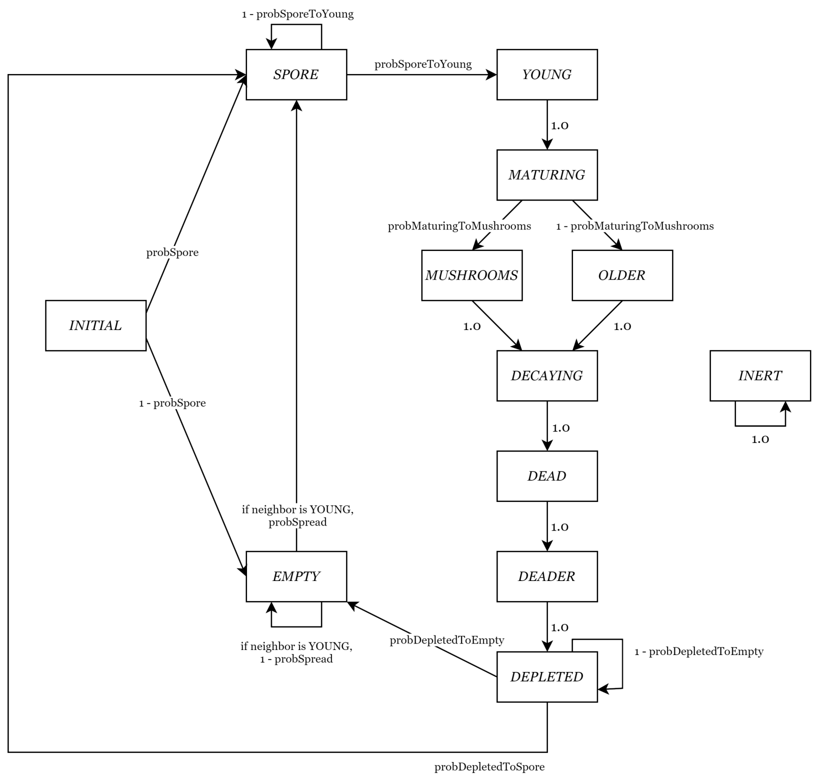
Table 2: Transition rules

current state	potential new state	transition probability
NULL	EMPTY	0.999
NULL	SPORE	0.001
EMPTY	YOUNG	0.6 (if $\geq$ 1 YOUNG neighbor)
SPORE	YOUNG	0.25
YOUNG	MATURING	1.0
MATURING	MUSHROOMS	0.7
MATURING	OLDER	0.3
MUSHROOMS	DECAYING	1.0
OLDER	DECAYING	1.0
DECAYING	DEAD	1.0
DEAD	DEADER	1.0
DEADER	DEPLETED	1.0
DEPLETED	EMPTY	0.5
DEPLETED	SPORE	0.0001

Table 3: Probability values

value	prob.	description
probSpore	0.001	NULL $\rightarrow$ SPORE
probSporeToYoung	0.25	SPORE $\rightarrow$ YOUNG
probSpread	0.6	EMPTY with $\geq$ 1 YOUNG neighbor $\rightarrow$ YOUNG
probMaturingToMushroom	0.7	MATURING $\rightarrow$ MUSHROOMS (else $\rightarrow$ OLDER)
probDepletedToSpore	0.0001	DEPLETED $\rightarrow$ SPORE
probDepletedToEmpty	0.5	DEPLETED $\rightarrow$ EMPTY

Figure 1: State diagram



pattern both sequentially and in parallel. The simulation is written entirely in C++ and compiled with G++, and can be accessed via terminal application. I ran the simulation on a multicore server provided by Macalester, which allowed me to test the program on up to 32 processors. The process management and general parallelization was done through the `OpenMP` API.[3]

I determined 11 possible states for each cell, as well as a rule set to determine transitions between states. The states are defined in table 1. Each cell is initialized as either `EMPTY` or `SPORE` to create the starting grid for the simulation; at each time step, depending on the contents of the cell and its neighbors, its value is determined at the next time step by a probability calculation, described in table 2. If the cell state fails the probability check and does not transition, then the cell has the same state in the next time step as in the current; the self-transitions have been omitted from the probability chart for the sake of clarity.

I then mapped a state diagram for the cell values based on the example diagrams from the source that inspired the project topic, *Introduction to Computational Science*,[1] shown in figure 1. Along with the coded probability values shown in table 3, I was able to fully implement the rule set for the

Figure 2: The collapse() function

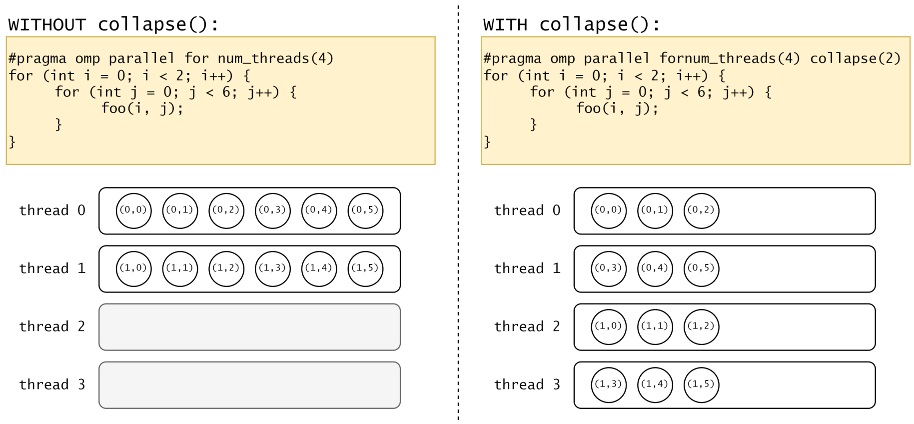
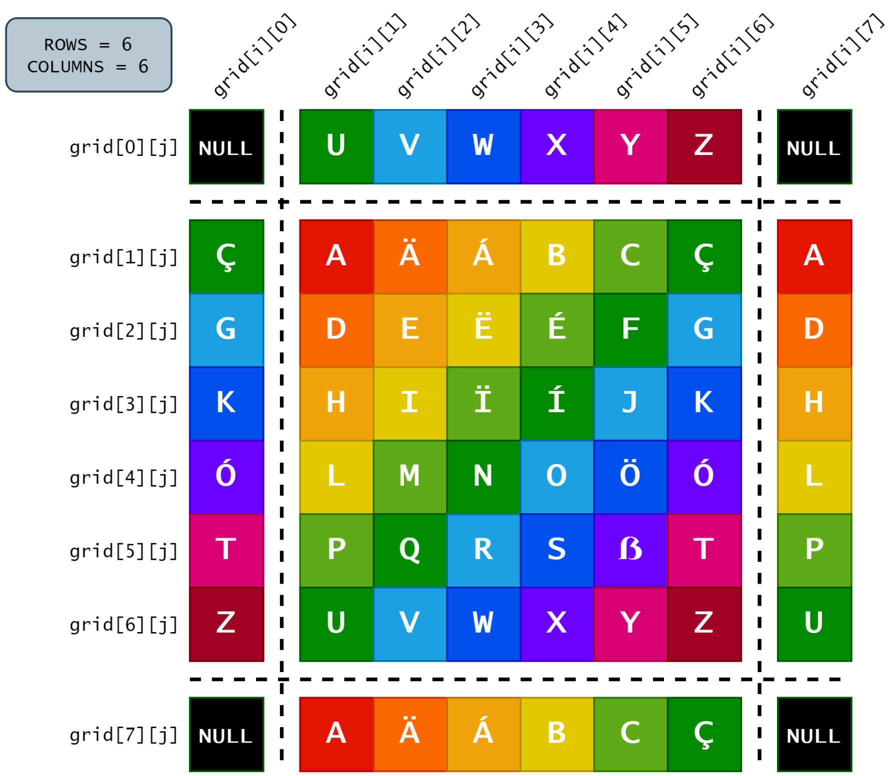


Figure 3: A visualization of the ghost cell strategy for structured grids



stochastic mechanism.

Note that in the sequential version, all of the arguments passed to the functions are done so by reference rather than by value to save memory space as well as to avoid artificial increases in runtime at higher problem sizes caused by repeatedly copying larger objects. In the parallel version, arguments passed to functions to serve as iterators in parallelized loops have instead been declared separately within the functions themselves; this is because `OpenMP` is able to automatically divide work done in a for-loop between threads, but requires that the loop iterator be private to each thread (typically designating the object automatically) and a dereferenced pointer is not recognized by the package to be the object stored at the address.[3]

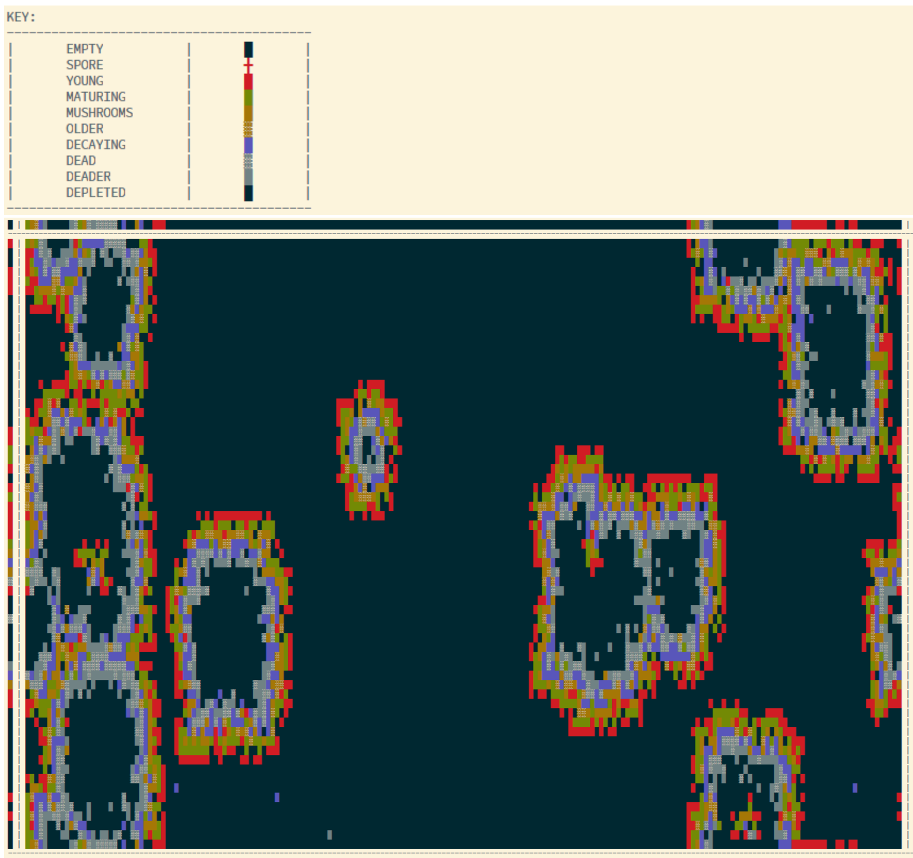
The first part of the solution is the declaration and initialization of the grid itself. Running the compiled program takes three arguments in the sequential version: the number of rows, the number of columns, and the number of time steps, adding the number of threads as a fourth argument in the parallel version. I used the built-in C function `getopt()` and a simple switch statement to parse the command line before checking the arguments were all in the acceptable range (see function `getArguments`, `fungi-seq.cpp` lines 122-189 and `fungi-omp.cpp` lines 145-229).

To allocate the grid, I opted to use

Figure 4: An example of the numerical grid terminal output after 15 time steps with 35 rows and 50 columns

```
6 | 4 5 6 7 7 8 9 0 0 9 0 0 0 0 0 9 0 0 0 9 9 7 7 4 5 2 3 3 3 2 3 3 2 0 3 3 6 6 0 6 6 6 2 6 6 4 6 6 6 6 | 4
-------------------------------------------------------------------------------------------------------
7 | 6 4 6 4 7 8 9 0 9 9 0 9 0 0 0 9 0 0 0 9 9 8 6 6 4 0 2 0 0 2 0 2 0 2 0 0 6 7 7 7 7 7 6 7 6 7 6 7 7 7 | 6
6 | 4 3 4 4 6 7 8 9 8 9 9 9 9 9 9 0 9 9 0 0 9 8 7 6 3 2 0 0 2 0 2 0 2 3 4 3 6 7 8 8 8 8 8 8 8 8 8 8 8 6 | 4
8 | 4 2 6 5 6 6 7 7 9 9 9 8 9 9 7 6 0 9 8 0 8 8 7 5 2 0 3 3 2 3 3 3 3 3 4 6 6 7 8 9 0 9 9 9 9 9 9 9 9 8 | 4
6 | 7 7 7 9 6 0 6 8 8 8 0 7 8 8 8 7 8 6 7 7 6 7 3 3 2 2 4 3 5 5 0 4 2 0 5 6 7 0 8 9 0 9 0 9 9 9 9 9 8 6 | 7
9 | 8 8 7 7 3 6 7 7 7 7 7 7 4 7 7 7 7 7 2 3 6 6 4 3 3 3 4 6 6 6 4 6 4 2 5 5 7 8 9 9 0 9 0 0 0 0 0 0 9 9 | 8
0 | 8 9 8 3 6 4 6 6 6 6 6 6 4 6 3 3 4 6 6 3 5 4 4 3 4 3 6 3 7 7 7 0 4 3 4 6 7 8 9 9 0 9 0 9 9 0 0 9 0 0 | 8
9 | 0 9 8 4 2 4 0 4 4 3 2 4 2 3 4 3 5 2 5 5 0 3 3 5 6 4 7 7 7 8 8 8 8 8 8 8 8 8 9 9 9 0 9 0 9 0 0 9 0 9 | 0
0 | 9 8 8 7 4 2 3 3 3 3 3 0 2 2 2 3 3 3 0 3 3 0 3 6 7 7 8 8 7 7 9 9 8 9 7 9 9 9 9 0 7 0 0 0 0 0 0 0 8 0 | 9
0 | 9 0 8 5 6 3 3 0 2 2 2 0 0 0 2 2 2 0 2 2 2 2 4 6 7 8 6 9 9 0 9 9 9 0 0 9 0 9 9 8 9 0 0 0 0 0 0 0 0 0 | 9
0 | 8 9 8 7 6 4 3 2 0 0 0 0 0 0 0 0 0 0 0 0 2 3 4 3 3 8 9 8 9 9 0 0 0 0 9 0 0 0 9 9 0 9 0 0 0 0 0 0 0 0 | 8
0 | 9 9 8 7 4 5 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 3 4 6 5 8 9 9 0 9 0 9 0 9 0 0 0 9 0 9 0 0 0 0 0 0 0 0 0 0 | 9
0 | 0 9 7 7 5 4 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 4 6 7 7 9 0 0 0 0 0 0 0 0 0 9 9 9 0 9 0 0 0 0 0 0 0 0 0 | 0
0 | 9 7 8 7 6 4 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 6 4 8 7 0 0 0 0 0 0 0 0 0 0 9 9 0 0 9 0 0 0 0 9 0 0 0 | 9
8 | 9 8 7 6 6 3 3 2 0 0 0 0 0 0 0 0 0 0 0 0 2 0 5 0 7 7 8 9 9 0 9 0 0 0 0 0 0 0 9 9 9 0 0 9 0 9 0 0 0 8 | 9
8 | 7 6 6 6 3 4 0 2 0 0 0 0 0 0 0 0 0 0 0 2 0 5 6 6 7 7 9 9 9 0 0 0 0 0 0 0 0 0 0 9 9 0 0 0 0 9 9 0 0 8 | 7
8 | 7 6 4 3 5 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 3 5 6 7 7 8 9 0 9 0 0 0 0 0 0 0 0 9 0 9 0 9 0 9 0 9 0 9 8 8 | 7
8 | 7 6 3 3 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2 4 4 6 8 8 9 0 9 9 0 0 0 9 0 0 0 0 9 9 9 7 9 7 6 9 7 9 9 8 | 7
8 | 4 6 4 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 5 8 9 0 0 9 0 0 9 0 9 0 0 0 9 0 7 8 8 8 8 4 8 8 8 8 8 | 4
7 | 7 4 4 3 3 3 3 0 2 2 2 2 2 2 0 0 2 2 2 0 3 0 5 7 8 9 9 9 0 0 0 0 0 9 0 9 9 0 0 9 8 9 7 7 7 7 7 7 7 7 | 7
6 | 6 6 3 3 3 5 2 2 3 3 2 2 3 3 3 3 3 3 3 3 3 5 0 6 8 9 9 9 0 0 0 0 0 0 9 0 0 0 0 9 0 8 8 7 6 4 6 4 6 6 | 6
2 | 4 4 4 5 6 4 3 0 3 4 4 4 4 4 4 4 4 4 5 4 2 5 6 7 8 9 0 0 9 0 0 0 0 0 0 9 0 0 0 0 9 8 8 7 3 5 0 2 3 2 | 4
3 | 3 2 0 2 6 7 3 0 3 4 6 6 5 6 6 6 0 4 6 6 6 4 6 4 8 9 9 9 0 0 0 0 0 0 9 0 0 0 0 8 9 9 8 6 6 2 0 2 2 3 | 3
2 | 3 2 3 6 5 4 8 8 8 8 8 7 7 7 6 7 7 6 7 7 3 3 6 6 8 9 9 0 0 0 0 0 0 0 0 0 0 0 0 0 9 9 8 6 5 4 0 0 0 2 | 3
0 | 0 5 4 4 7 7 6 9 9 8 9 8 8 7 5 8 8 8 8 7 6 4 6 7 7 9 0 0 9 0 0 0 0 0 0 0 0 9 9 0 9 8 7 7 5 0 3 2 0 0 | 0
2 | 3 3 6 6 7 8 8 7 9 9 9 8 8 9 9 9 8 9 8 5 4 3 4 6 8 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 9 7 7 6 5 3 0 0 2 | 3
2 | 2 2 6 7 7 0 9 7 9 9 9 9 9 0 0 9 0 9 8 3 5 2 4 7 7 9 9 0 0 0 0 0 0 0 0 0 9 0 9 9 0 9 8 7 3 4 3 2 0 2 | 2
0 | 3 3 4 7 8 8 9 0 0 0 0 9 0 9 0 0 9 7 7 7 7 6 6 7 8 9 0 9 0 0 0 0 9 0 0 0 9 9 0 9 0 8 7 7 3 4 3 2 0 0 | 3
2 | 3 4 6 7 8 9 8 9 0 0 0 2 2 0 0 0 9 6 5 8 7 6 7 7 7 9 7 9 9 9 0 0 0 9 0 0 8 9 0 9 9 9 8 6 6 5 3 0 0 2 | 3
2 | 3 5 4 3 8 9 9 0 0 0 2 3 0 0 0 0 9 5 9 9 8 8 0 7 8 8 9 6 9 9 9 9 8 9 9 0 9 9 9 9 8 6 3 0 2 3 3 2 0 2 | 3
0 | 3 3 3 0 8 9 0 0 0 0 0 4 0 0 2 9 2 0 0 9 9 8 6 7 5 8 7 8 9 0 9 9 8 8 0 9 7 7 7 8 7 3 4 3 0 2 2 2 0 0 | 3
0 | 3 4 3 4 8 9 8 9 0 4 6 0 0 9 9 3 2 2 9 9 8 4 7 6 7 6 7 8 8 8 8 8 8 0 6 8 8 7 7 7 7 6 4 3 0 0 0 0 0 0 | 3
0 | 3 3 6 6 8 9 0 9 0 3 6 7 8 8 5 4 3 0 0 0 9 6 7 4 6 5 6 6 7 7 7 7 7 7 6 7 5 7 6 4 2 6 0 0 0 2 2 2 2 0 | 3
0 | 3 2 6 7 8 8 9 0 0 2 3 4 0 7 6 0 3 2 9 0 9 8 7 6 4 4 4 0 6 6 6 6 6 0 6 4 4 6 5 3 3 4 3 3 3 0 3 3 3 0 | 3
3 | 4 4 4 3 8 9 9 0 0 2 3 3 9 9 0 2 0 2 0 9 9 8 6 6 4 3 3 5 3 4 4 2 2 2 3 4 5 4 5 4 4 2 2 3 4 5 0 2 5 3 | 4
6 | 4 5 6 7 7 8 9 0 0 9 0 0 0 0 0 9 0 0 0 9 9 7 7 4 5 2 3 3 3 2 3 3 2 0 3 3 6 6 0 6 6 6 2 6 6 4 6 6 6 6 | 4
-------------------------------------------------------------------------------------------------------
7 | 6 4 6 4 7 8 9 0 9 9 0 9 0 0 0 9 0 0 0 9 9 8 6 6 4 0 2 0 0 2 0 2 0 2 0 0 6 7 7 7 7 7 6 7 6 7 6 7 7 7 | 6
```

Figure 5: An example of the color-coded grid terminal output after 15 time steps with 65 rows and 200 columns



the `new` command instead of `malloc` because I wanted to parallelize the allocation to avoid slowdown at higher grid sizes, and `malloc` requires additional functions to work within `OpenMP` whereas `new` works as it normally would in a sequential environment.[4] I created two 2D pointer arrays to hold the grid at the current time step and at the next time step, called `current_grid` and `next_grid` respectively, so that the processing during each step could happen concurrently and therefore improve the potential for task partitioning.[2] I initialized both arrays as 1D pointer arrays with (number of rows + 2) elements, then used a parallel for-loop to create new 1D pointer arrays with (number of columns + 2) elements at each index of the container array (see function `allocateGrid`, `fungi-seq.cpp` lines 193-198 and `fungi-omp.cpp` lines 233-239).

To initialize the grid, I used a doubly-nested for-loop to parse through every cell in the grid and perform a stochastic step to determine whether each cell was to start as `EMPTY` or as `SPORE`. The `OpenMP` parallel for-loop automatically divides work among the threads by the outermost loop only in the case of nesting, and a common solution is to modify the nested loops to create an equivalent multi-iterator loop (function `initializeGrid`, `fungi-seq.cpp` lines 202-213 and `fungi-omp.cpp` lines 243-256). For the sake of clarity and to ensure I could still account for a potential number of rows (iterated by the outer loop) that would not utilize all of the threads, I included the `OpenMP` command `collapse()` to automatically distribute work for the nested loops without manually combining them.[5] An explanation of this function can be seen in figure 2.

To determine whether a cell would begin as `EMPTY` or `SPORE` in the start state, I generated a decimal number between 0 and 1 and checked whether this number fell between 0 and `probSpore` = 0.001. If the generated number did fall within this range, the cell was made a `SPORE`, and otherwise it was made `EMPTY`. For the RNG, I chose to use *Tina's Random Number Generator* (TRNG), which allows for an RNG engine to be seeded and split among threads; this is necessary because a standard sequential-only RNG (such as the built-in `rand()` function in C++) creates a list of random values and returns the next value each time a new pseudo-random number is called for, and attempting to access this structure with multiple processes will cause compilation and runtime errors at best and logic errors at worst.[6] Within TRNG, I chose the `yarn2` engine and the `uniform01` distribution function to create the necessary decimal values with an equal likelihood of each point in the range to best simulate the outcome of the probability calculations. After the grids were initialized in the start state, the simulation itself could begin (see function `mushrooms`, `fungi-seq.cpp` lines 217-351 and `fungi-omp.cpp` lines 260-397). The outermost for-loop iterates the simulation over each time step; the processing for each time step depends on the outcome of the step before it, requiring this loop to be processed sequentially. This rigidly sequential structure is one of the major limitations on potential benefits of parallelizing this problem.

At the start of each time step, a series of *ghost cells* are established. Ghost cells (also called *ghost nodes*) are a tool in structured grid appriaches to deal with border grid points that require information about their neighbors to determine their updated state. In practice, ghost cells are simply duplicated data to minimize unnecessary communication between threads. In this case, the bottom, top, leftmost, and rightmost rows of the working grid are copied to the top, bottom, rightmost, and leftmost rows of the 2D array, respectively. This is why the 2D array was initialized to have two more rows and columns than were being used for the grid area itself. A visualization of this technique can be seen in figure 3. The values of the ghost rows are not computed directly and are not included in the workload of any thread, but copying their values across the edges as shown allows for edge cells to check the values of their neighbors without refreshing the cache or needing to get the value from another thread during the computation step; at the start of every step, the values in the working grid have all finished being updated and the ghost cells can then be updated to match.

After updating the ghost cells, every working cell in `current_grid` is visited and its value read before determining if a change of state will occur. To visit each cell, a doubly-nested for-loop checking every combination of row and column values uses the same `collapse` structure as described previously and explained in figure 2. When a cell is read, the value enters a switch statement to determine what rule(s) may apply to the cell before checking the associated probability operation in the same manner as discussed for initialization with varying probability thresholds. Certain state transitions have a probability of 1.0 to occur, meaning that the cell will always update along that transition during a change in time step (i.e. `OLDER` → `DECAYING`).

When a given cell's state is determined in the next time step, the cell in `current_grid` is not modified; rather, the cell at the same indices (same row and same column) in `next_grid` is set to the updated value. After all of the cells in `current_grid` have been processed, the values in `next_grid` are read and written back into `current_grid` so that the next time step calculation may begin (if it is not the last time step). This is done with a doubly-nested for-loop run in parallel with the `collapse()` function as previously described. In each step, after the ghost cells have been copied and before the processing of cell states begins, a visualization of the grid is displayed if the appropriate flags have been set; the function call is placed here because the ghost cell copies are practically the final step of the processing for the previous time step and they should be updated prior to displaying the grid values, but because the ghost cell copies are not necessary in the final time step they occur at the start of the loop, necessitating the grid printing to occur in the middle. The default display of the grid is numerical, with the integer value for each cell being read and printed to spatially reflect the indices of the cell (see function `print_number_grid`, `fungi-seq.cpp` lines 389-427 and `fungi-omp.cpp` lines 444-482); an example is shown in figure 4 with dotted lines separating the working grid from the ghost cells. If the color-coded grid is enabled, a block corresponding to the value of the cell is printed in place of the number, making it easier to identify the growth patterns (see function `print_colorful_grid`, `fungi-seq.cpp` lines 431-601 and `fungi-omp.cpp` lines 486-655); an example is shown in figure 5 with dotted lines separating the working grid from the ghost cells. To implement this grid, I made use of several commands in the built-in `printf` function in C++ to print characters by their Unicode decimal identifier and to set the color of the printed text (see functions `reset_color`, `black`, `red`, `green`, `brown`, `grey`, and `purple`, `fungi-seq.cpp` lines 605-643 and `fungi-omp.cpp` lines 659-697).[7] In addition to being easier to understand as a data visualization, the color-coded grid is able to display larger grids in the same amount of space, as the numbers must be a certain size to remain readable where the block characters can be much smaller.

During the processing of the cell states, one rule requires the neighbors of the cell to be checked: `EMPTY` → `YOUNG` requires that a probability operation be checked if at least one neighbor of the `EMPTY` cell is `YOUNG`. This is essentially a boolean operation determining whether `neighbor1` $\vee$ `neighbor2` $\vee \ldots \vee$ `neighbor7` $\vee$ `neighbor8 = TRUE` where each neighbor is `TRUE` if it contains the value `YOUNG` and `FALSE` otherwise. Ordinarily this would be done with a loop checking the cells neighbors and a return statement if one neighbor is found to be true, and this is how it is implemented in the sequential version, however parallel processes cannot break out of a parallel loop based on a condition determined at runtime. To produce the same result, I used an arithmetic `reduction` loop from `OpenMP` to create an arbitrary holder `int` called `young` and allow each thread to increase the value of `young` by 1 each time it finds a `YOUNG` neighbor cell (see function `check_neighbors`, `fungi-seq.cpp` lines 365-376 and `fungi-omp.cpp` lines 412-430). After the loop is completed, I simply check if `young != 0`, returning 1 (representing `TRUE`) if it is and 0 (representing `FALSE`) otherwise. The `check_neighbors` function is called from inside the switch statement in the `mushrooms` function, and based on the value returned either performs the stochastic step to determine if the cell will become `YOUNG` or simply leave the cell as `EMPTY` in the next time step.

After the processing is finished, the runtime for the simulation is calculated and printed to the terminal and the arrays must be deallocated. Because C is not a memory safe language and the 2D arrays were allocated using pointers, they need to be manually deleted. To do this, I iterate through each container array and delete the pointer arrays at each element before deleting the container array (see function `deallocateGrid`, `fungi-seq.cpp` lines 380-385 and `fungi-omp.cpp` lines 434-440). After deallocating both `current_grid` and `next_grid`, the simulation ends.

2.2 Running the simulation

The repository containing the simulation files is publically available at `github.com/aronsmithdonovan/ParallelizationCapstone`; download the repository, open the local folder in a text editor, and navigate into the main directory using a terminal application.

Open the `Makefile` inside the main directory and ensure the desired flags are set:

- enable `DEBUG` to print a numerical representation of the grid at each time step to the terminal
- with `DEBUG` enabled, enable `COLOR` to change the numerical grid to a color-coded grid of blocks (recommended for clarity)
- for generating data, it is advised to disable both flags

Execute the following command inside the main directory to create the sequential processing file:

```
$make seq.fungi
```

Execute the following command to run the simulation sequentially:

```
$./seq.fungi -r R -c C -s S
```

where `R` is the number of rows, `C` is the number of columns, `S` is the number of time steps, and `R`, `C`, and `S` are all be positive nonzero integers; an error will be thrown at runtime if the arguments supplied do not meet this criteria. Execute the following command inside the main directory to create the parallel processing file:

```
$make omp.fungi
```

Execute the following command to run the simulation in parallel:

```
$./omp.fungi -r R -c C -s S -t T
```

where `R` is the number of rows, `C` is the number of columns, `S` is the number of time steps, `T` is the number of threads, and `R`, `C`, `S`, and `T` are all be positive nonzero integers; an error will be thrown at runtime if the arguments supplied do not meet this criteria.

3 Assessment

3.1 Methodology

To assess the performance improvement of the parallelized program, I chose to investigate both strong and weak runtime scalability. *Amdahl's law* says that increasing the number of processes working on a problem gives diminishing returns on *speedup*.[8] Speedup can be defined as $t(1)/t(n)$ where $t(1)$ is the total time to run the program with one processor and $t(n)$ is the total time to run the program with N processors.[9] Ideally speedup will be equal to N, but by Amdahl's law we know that speedup cannot be linear at high enough values of N. Strong scalability testing calculates the speedup value for different numbers of processes at the same problem size; I have repeated the strong scalability calculations for multiple problem sizes to be as thorough as possible. *Gustafson's law*, on the other hand, says that increasing the number of processes working on a problem gives proportionate returns on scaling.[8] Weak scalability compares runtimes for different problem sizes with different numbers of processes with a scalar maintained between both values, e.g. testing 2 threads on a problem size of 10, then 4 threads on a problem size of 20, then 8 threads on a problem size of 40, etc. Because the problem size for the structured grid simulation is affected by the number of rows, the number of columns, and the number of time steps, I chose to keep the number of time steps constant at 100 and to keep the number of rows and columns equal; the problem size will henceforth be referred to by the number of rows and columns, understanding that the size of the grid will be equal to this value squared. For my strong scalability testing, I have generated median runtime data for 1, 2, 4, 8, 16, and 32 processes for problem sizes of 100, 250, 425, 600, 850, 1000 and 1200 with 100 time steps for every test. For my weak scalability testing, I have generated median runtime data for 2, 4, 8, and 16 processes with a starting problem size of 100 and 4 scalar repetitions. The problem sizes have been selected such that the grid itself roughly doubles inside for each iteration of weak testing.

3.2 Findings

Figure 6: Median runtimes at different problem sizes for 1 thread

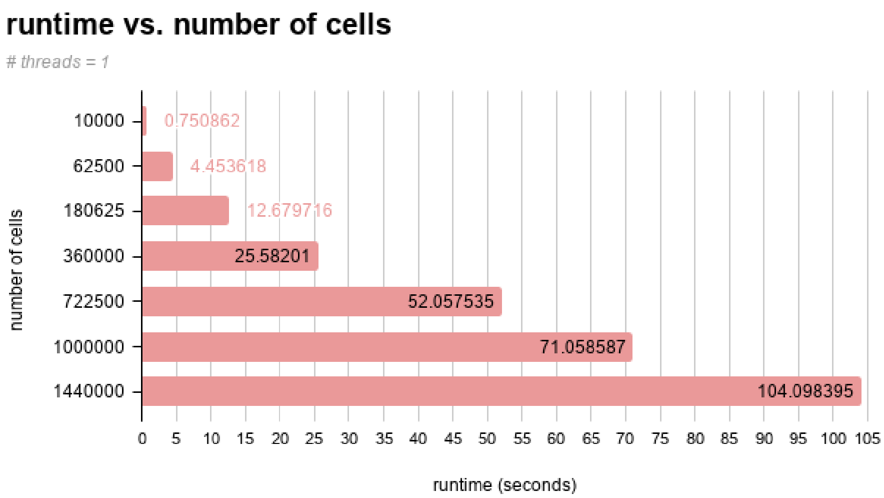


Figure 7: Median runtimes at different problem sizes for 2 threads

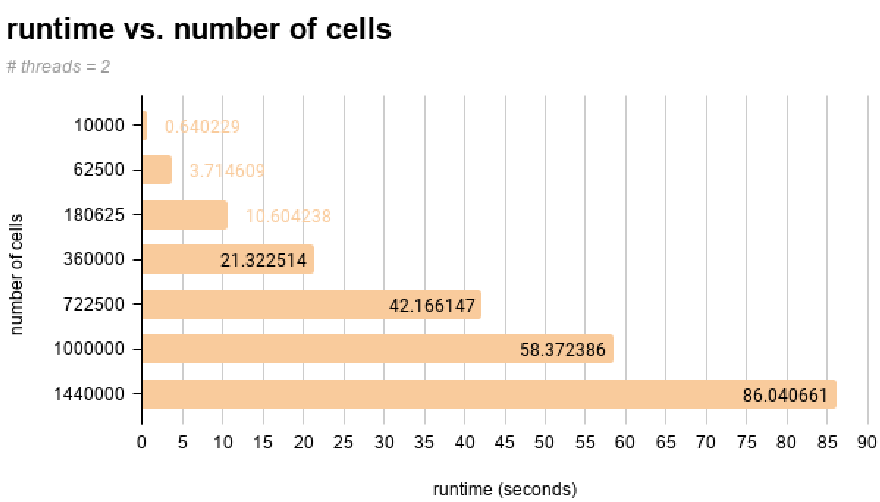


Figure 8: Median runtimes at different problem sizes for 4 threads

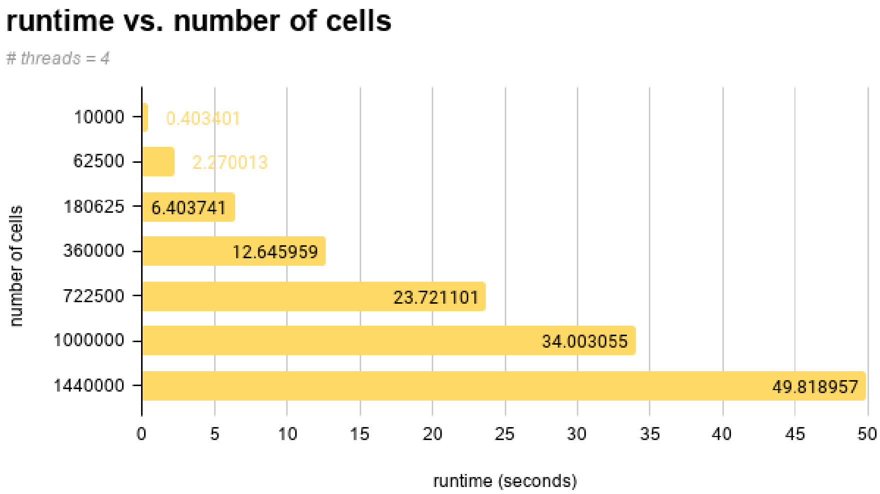


Figure 9: Median runtimes at different problem sizes for 8 threads

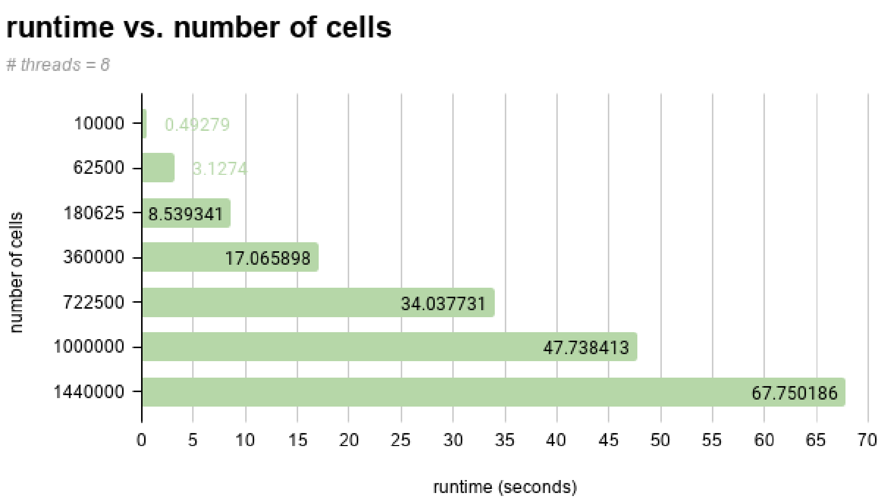


Figure 10: Median runtimes at different problem sizes for 16 threads

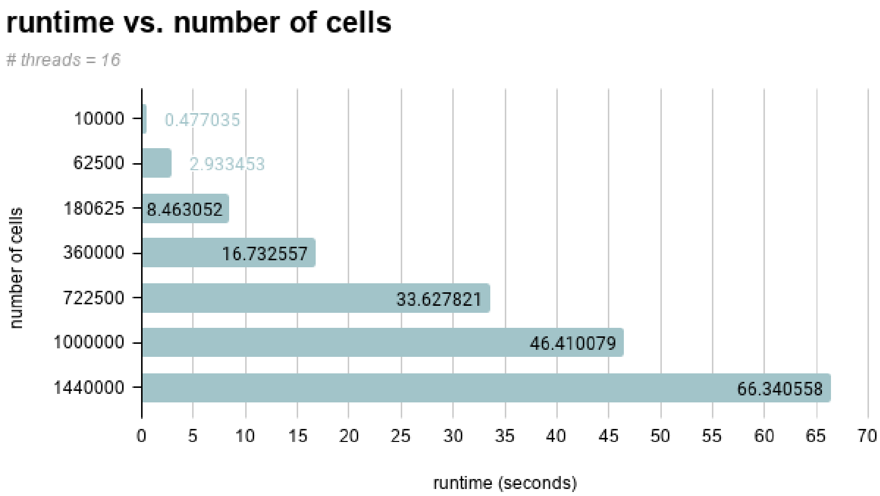


Figure 11: Strong scalability speedup results

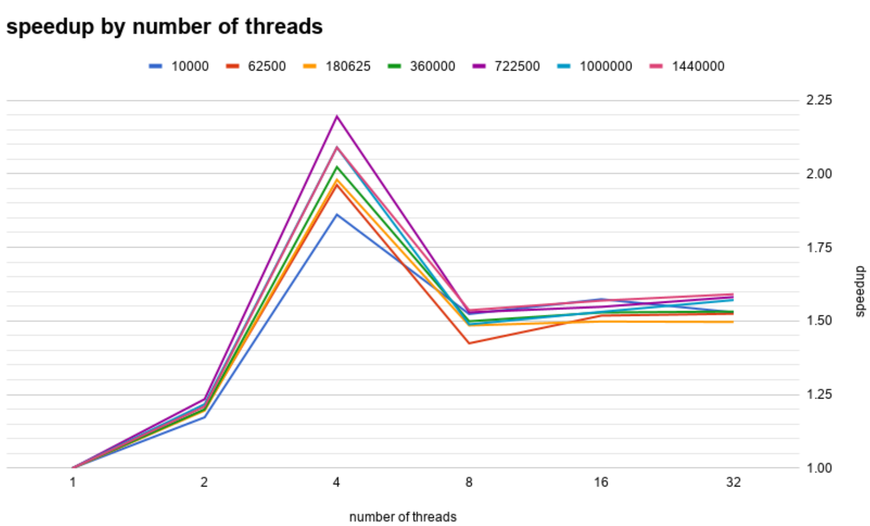
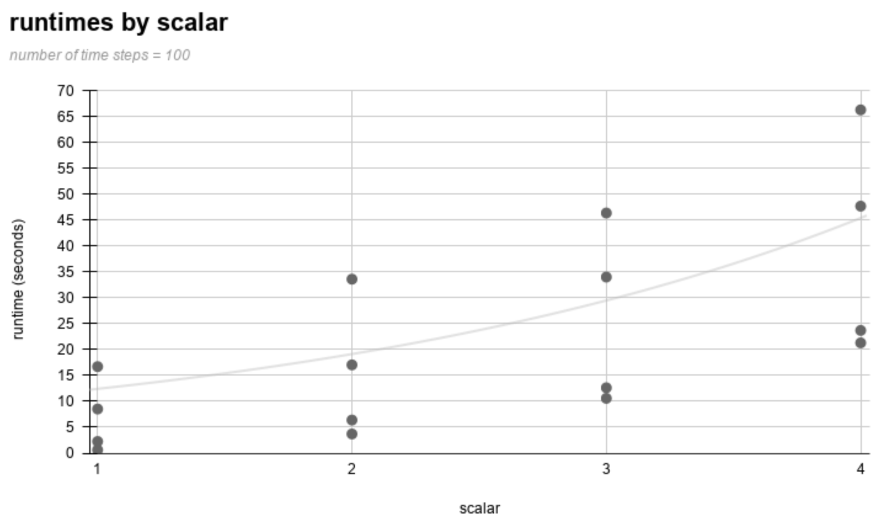


Figure 12: Weak scalability scalar runtime results



3.3 Discussion of results

Based on the visualization in figure 11, it is clear that the parallel program has poor strong scalability, especially for more than 4 processes. The ideal speedup at a given number of processes is equal to the number of processes, and this ideal circumstance is not achieved at any point. Ordinarily a line for the ideal speedup would be shown on the graph for comparison, but this affects the scale significantly and makes it difficult to read the rest of the graphic, so it has been excluded here.

This outcome reinforces Amdahl's law, as we have shown the improvement on runtime diminishing as we increase the number of threads. This law is based on the idea that some part of the program must be running sequentially, and as the number of threads increases, the time taken to perform the sequential steps limits any further decrease in runtime;[8] in this problem, as previously mentioned, each time step for the grid must be computed sequentially because it depends on the outcome of the step before it. From this we can reasonably assume that the main cause of the diminishing speedup values is this sequential restriction.

Looking now to figure 12, the parallel program seems to have moderate weak scalability: the runtimes by scalar are more spread out than the ideal circumstance, but the overall trend is still clearly exponential. Models of different sizes are being generated within a reasonable range of runtimes at each iteration by scaling the number of processes by the problem size, which aligns with our expectations based on Gustafson's law. There is not a strong enough trend here to declare good weak scalability, but these results indicate that scaling the number of processes to meet increasing problem sizes does bring the runtime within an expected range.

Overall, we can conclusively say that the program lacks strong scalability as indicated by the data collected. In addition, although it is not yet determined to what degree the program is weakly scalable, it has not yet been shown to conclusively lack weak scalability; as such, further experimentation is needed to finalize a claim.

4 Future Steps

4.1 Expanding results

The weak scalability data gathered intentionally stopped at a problem size of 1200; this is because the runtimes for bigger problem sizes became increasingly long, and it was not possible to capture enough data points without the server automatically terminating the connection. These results must be acknowledged to be inconclusive; however, because a potential trend is indicated, a crucial step were the project to continue would be to generate data at these larger problem sizes to draw a firm conclusion.

4.2 Improving parallelization

The process of parallelizing a problem requires striking a balance of *task granularity*: the problem must be broken into smaller pieces to be performed concurrently, but excessively small tasks require so much overhead to partition that more efficiency is lost than gained in the process. For this parallelization, I kept the sequential time steps and parallelized the grid access iterations; a potential future direction for the project would be to revisit the sequential steps in search of parallelizable components and to attempt different approaches to the decomposition of the problem into tasks, ultimately looking for improvements in scalability as a results of these changes.

The `OpenMP` package is one of several potential strategies to control parallel processes (e.g. `OpenACC` or `MPI`), and so another possible space for future improvements would be to re-work the program with other available controllers that may allow for different methods of task partitioning than were possible with `OpenMP`.

4.3 Grid variations

The problem description that inspired this project suggested the possibility of an `INERT` state for cells which could only stay `INERT` on each grid update step. However, there is no obvious way that a cell would become inert as a result of the fungal growth process being modeled, and so because an additional factor or factors would need to be considered the `INERT` state was implemented with no path to reach it. In its place, I developed the `DEPLETED` state to serve as a buffer between fungi dying from lack of nutrients and new fungi growing in the same location.

A potential future direction here would not be to improve the parallelization, but to improve the problem approach; an `INERT` cell could be declared as such in the initialization of the grid, which would allow us to simulate the growth pattern with certain areas of the grid not being suitable for fungal growth, e.g. areas with rocks or other plant growth.

4.4 Improving cell states

As an extension of fully implementing the `INERT` state, another future task is to develop additional states to more accurately simulate the fungal growth cycle. Although the fairy ring pattern is clear in the current algorithm, there are other factors that could be accounted for if I were to continue working on the project, e.g. soil quality, weather, interactions with other fungi, animal interactions, and spore scattering patterns. Ultimately, every model will fail to accurately represent some component of the actual phenomena, but there is clear room for improvement in the variety of conditions recognized as the mycelium network expands and develops.

Works Cited

[1] Angela B. Shiflet and George W. Shiflet. *Introduction to Computational Science: Modeling and Simulation for the Sciences*. 2nd ed. Princeton University Press, 2014. ISBN: 9780691160719.

[2] Katya Gonina and Hovig Bayandorian. *Structured Grids*. URL: https://patterns.eecs.berkeley.edu/?page_id=498.

[3] *OpenMP Application Programming Interface*. 2018. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[4] *Dynamic memory management in target regions*. 2019. URL: https://downloads.ti.com/mctools/esd/docs/openmpacc/dynamic_memory_management.html.

[5] Jukka Suomela, Samuli Laine, and Jaakko Lehtinen. *Multithreading with OpenMP*. 2020. URL: https://ppc.cs.aalto.fi/ch3/nested/.

[6] Heiko Bauke. *function rand ¡cstdlib¿*. 2020. URL: https://www.cplusplus.com/reference/cstdlib/rand/.

[7] *The Unicode Standard: Block Elements*. Version 13. 2020. URL: https://unicode.org/charts/PDF/U2580.pdf.

[8] Dr. Taek Kwon. *Multiprocessor Laws*. 2020. URL: https://www.d.umn.edu/~tkwon/course/5315/HW/MultiprocessorLaws.pdf.

[9] *Scaling Tutorial*. 2020. URL: https://hpc-wiki.info/hpc/Scaling_tutorial.

Source Code

fungi-seq.cpp:

```cpp
/*******************************************************************************************
 * fungi-seq.cpp
 *******************************************************************************************
 *
 * sequentially simulates the growth of a mushroom network in a patch of grass
 *
 * created by Aron Smith-Donovan using code written by Libby Shoop as reference
 *
 * based on a project description posited in "Introduction to Computational Science:
 *      Modeling and Simulating for the Sciences" by Angela B. Shiflet and George W Shiflet
 *
 *
*/

/* LIBRARIES */
    #include <stdlib.h>
    #include <stdio.h>
    #include <string.h>
    #include <unistd.h>
    #include <cstdlib>
    #include <iostream>
    #include <trng/yarn2.hpp>
    #include <trng/uniform01_dist.hpp>
    #include <locale.h>
    #include <wchar.h>
    #include "seq_time.h"  // Libby's timing function that is similar to omp style

/* UNIVERSAL CONSTANTS */
    // probability values for state changes
    #define probSpore 0.001             // probability that a site initially is SPORE
    #define probSporeToYoung 0.25       // probability that a SPORE will become YOUNG at the next time step
    #define probSpread 0.6              // probability that a EMPTY with a neighbor that is YOUNG will become YOUNG at the next time step
    #define probMaturingToMushrooms 0.7 // probability that a MATURING will become MUSHROOMS at the next time step (otherwise it becomes OLDER)
    #define probDepletedToSpore 0.0001  // probability that a DEPLETED will become SPORE at the next time step
    #define probDepletedToEmpty 0.5     // probability that a DEPLETED will become EMPTY at the next time step

    // cell states
    #define EMPTY 0      // empty ground containing no spore or hyphae
    #define SPORE 1      // contains at least one spore
    #define YOUNG 2      // young hyphae that cannot form mushrooms yet
    #define MATURING 3   // maturing hyphae that cannot form mushrooms yet
    #define MUSHROOMS 4  // older hyphae with mushrooms
    #define OLDER 5      // older hyphae with no mushrooms
    #define DECAYING 6   // decaying hyphae with exhausted nutrients
    #define DEAD 7       // newly dead hyphae with exhausted nutrients
    #define DEADER 8     // hyphae that have been dead for a while
    #define DEPLETED 9   // area whose nutrients have previously been depleted by fungal growth
    #define INERT 10     // inert area where plants cannot grow

/* FUNCTION DECLARATIONS */
void getArguments(int argc, char *argv[], int * ROWS, int * COLUMNS, int * TIME_STEPS);
void allocateGrid(int ***grid, int * ROWS, int * COLUMNS, int * current_row);
void initializeGrid(int ***grid, int * ROWS, int * COLUMNS, int * current_row, int * current_column, double * prob, trng::yarn2 * yarn, trng::uniform01_dist<> * uniform);
void mushrooms(int ***current_grid, int ***next_grid, int * ROWS, int * COLUMNS, int * TIME_STEPS, int * current_row, int * current_column, int * current_time_step, int * neighbor_row, int * neighbor_column, int * current_value, double * prob, trng::yarn2 * yarn, trng::uniform01_dist<> * uniform);
void copyGrid(int ***current_grid, int ***next_grid, int * ROWS, int * COLUMNS, int * current_row, int * current_column);
int check_neighbors(int ***current_grid, int * current_row, int * current_column, int * neighbor_row, int * neighbor_column);
void deallocateGrid(int ***grid, int * ROWS, int * current_row);
void print_number_grid(int ***grid, int * ROWS, int * COLUMNS, int * current_row, int * current_column);
void print_colorful_grid(int ***grid, int * ROWS, int * COLUMNS, int * current_row, int * current_column, int * current_value);
void reset_color();
void black();
void red();
void green();
void brown();
void grey();
void purple();

/* main */
int main(int argc, char **argv){

    // declare variables
    double start_time, end_time, total_time;  // hold timer values
    int ROWS, COLUMNS, TIME_STEPS;  // hold command line arguments
    int **current_grid;  // grid at current time step
    int **next_grid;  // grid at next time step
    int current_row, current_column;  // grid cell counters
    int current_time_step;  // time step counter
    int neighbor_row, neighbor_column;  // check_neighbors() counters
    int current_value;  // hold grid print values
```

```
    double prob;  // stores randomly generated probability values

    // initialize random number engine
    trng::yarn2 yarn;  // create engine object
    trng::uniform01_dist<> uniform;  // create distribution fxn

    // parse command line arguments
    getArguments(argc, argv, &ROWS, &COLUMNS, &TIME_STEPS);

    // start timing
    start_time = c_get_wtime();

    // allocate grids
    allocateGrid(&current_grid, &ROWS, &COLUMNS, &current_row);
    allocateGrid(&next_grid, &ROWS, &COLUMNS, &current_row);

    // initialize current_grid
    initializeGrid(&current_grid, &ROWS, &COLUMNS, &current_row, &current_column, &prob, &yarn, &uniform);

    // run the simulation
    mushrooms(&current_grid, &next_grid, &ROWS, &COLUMNS, &TIME_STEPS, &current_row, &current_column, &current_time_step, &neighbor_row, &neighbor_column, &current_value, &prob, &yarn, &uniform);

    // end timing and print result
    end_time = c_get_wtime();
    total_time = end_time - start_time;
    #ifdef DEBUG
        printf("\nruntime: %f seconds\n", total_time);
    #else
        printf("%f", total_time);
    #endif

    // deallocate grids
    deallocateGrid(&current_grid, &ROWS, &current_row);
    deallocateGrid(&next_grid, &ROWS, &current_row);

    // return statement
    return 0;

}

/* getArguments() */
/* fetches and stores command line arguments for # of rows, columns, and time steps */
void getArguments(int argc, char *argv[], int * ROWS, int * COLUMNS, int * TIME_STEPS) {

    // declare + initialize variables
    int c;
    int rflag = 0;
    int cflag = 0;
    int sflag = 0;

    // retrieve command line arguments
    while ((c = getopt (argc, argv, "r:c:s:")) != -1) {
        switch (c) {
            case 'r':
                rflag = 1;
                *ROWS = atoi(optarg);
                break;

            case 'c':
                cflag = 1;
                *COLUMNS = atoi(optarg);
                break;

            case 's':
                sflag = 1;
                *TIME_STEPS = atoi(optarg);
                break;

            case '?':
                if (optopt == 'r') {
                    fprintf (stderr, "Option -%c requires an argument.\n", optopt);
                } else if (optopt == 'c') {
                    fprintf (stderr, "Option -%c requires an argument.\n", optopt);
                } else if (optopt == 's') {
                    fprintf (stderr, "Option -%c requires an argument.\n", optopt);
                } else if (isprint (optopt)) {
                    fprintf (stderr, "Unknown option `-%c'.\n", optopt);
                } else {
                    fprintf (stderr, "Unknown option character `\\x%x'.\n", optopt);
                    exit(EXIT_FAILURE);
                }
        }
    }

    // check command line arguments
    if (rflag == 0) {
        fprintf(stderr, "Usage: %s -r number of rows\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (*ROWS < 1) {
```

```
        fprintf(stderr, "Usage: %s -r number of rows must be a positive nonzero integer\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (cflag == 0) {
        fprintf(stderr, "Usage: %s -c number of columns\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (*COLUMNS < 1) {
        fprintf(stderr, "Usage: %s -c number of columns must be a positive nonzero integer\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (sflag == 0) {
        fprintf(stderr, "Usage: %s -s number of time steps\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (*TIME_STEPS < 1) {
        fprintf(stderr, "Usage: %s -s number of time steps must be a positive nonzero integer\n", argv[0]);
        exit(EXIT_FAILURE);
    }
}

/* allocateGrid() */
/* allocates enough space for the input grid to store the rows and columns for the problem */
void allocateGrid(int ***grid, int * ROWS, int * COLUMNS, int * current_row) {
    *grid = new int*[(*ROWS) + 2];  // create pointer array
    for ((*current_row) = 0; (*current_row) <= (*ROWS) + 1; (*current_row)++) {  // at each element in the pointer array...
        (*grid)[*current_row] = new int[(*COLUMNS) + 2];  // ...create another pointer array
    }
}

/* initializeGrid() */
/* initializes the grid with empty spaces and spore spaces to begin the simulation */
void initializeGrid(int ***grid, int * ROWS, int * COLUMNS, int * current_row, int * current_column, double * prob, trng::yarn2 * yarn, trng::uniform01_dist<> * uniform) {
    for ((*current_row) = 1; (*current_row) <= (*ROWS); (*current_row)++) {  // for each row in the grid...
        for ((*current_column) = 1; (*current_column) <= (*COLUMNS); (*current_column)++) {  // for each cell in that row...
            (*prob) = (*uniform)(*yarn);  // get random double between 0 and 1
            if ((*prob) <= probSpore) {  // if prob is less than or equal to probSpore...
                (*grid)[*current_row][*current_column] = SPORE;  // ...then cell starts as SPORE
            } else {  // otherwise...
                (*grid)[*current_row][*current_column] = EMPTY;  // ...cell starts as EMPTY
            }
        }
    }
}

/* mushrooms() */
/* simulates the growth of mushroom networks into fairy rings */
void mushrooms(int ***current_grid, int ***next_grid, int * ROWS, int * COLUMNS, int * TIME_STEPS, int * current_row, int * current_column, int * current_time_step, int * neighbor_row, int * neighbor_column, int * current_value, double * prob, trng::yarn2 * yarn, trng::uniform01_dist<> * uniform) {
    for((*current_time_step) = 0; (*current_time_step) <= (*TIME_STEPS); (*current_time_step)++) {  // for each time step...

        // set up ghost rows
        for ((*current_column) = 0; (*current_column) <= (*COLUMNS) + 1; (*current_column)++) {

            // set first row of grid to be the ghost of the second-to-last row
            (*current_grid)[0][*current_column] = (*current_grid)[(*ROWS)][*current_column];

            // set last row of grid to be the ghost of the second row
            (*current_grid)[(*ROWS) + 1][*current_column] = (*current_grid)[1][*current_column];
        }

        // set up ghost columns
        for ((*current_row) = 0; (*current_row) <= (*ROWS) + 1; (*current_row)++) {

            // set left-most column to be the ghost of the second-farthest-right column
            (*current_grid)[*current_row][0] = (*current_grid)[*current_row][*COLUMNS];

            // set right-most column to be the ghost of the second-farthest-left column
            (*current_grid)[*current_row][(*COLUMNS) + 1] = (*current_grid)[*current_row][1];
        }

        // DEBUG: display current grid
        #ifdef DEBUG
            #ifdef COLOR
                setlocale(LC_ALL, "");
                printf("\ntime step %d:\n", (*current_time_step));
                print_colorful_grid(current_grid, ROWS, COLUMNS, current_row, current_column, current_value);
            #else
                printf("\ntime step %d:\n", (*current_time_step));
                print_number_grid(current_grid, ROWS, COLUMNS, current_row, current_column);
            #endif
        #endif

        // determine grid at next time step
        for ((*current_row) = 1; (*current_row) <= (*ROWS); (*current_row)++) {  // for each row in the grid...
            for ((*current_column) = 1; (*current_column) <= (*COLUMNS); (*current_column)++) {  // for each cell in that row...

                (*current_value) = (*current_grid)[*current_row][*current_column];

```

```
            switch(*current_value) {

                // if current cell is EMPTY...
                case 0:
                    if (check_neighbors(current_grid, current_row, current_column, neighbor_row, neighbor_column) == 0) {  // if cell has no YOUNG neighbors...
                        (*next_grid)[*current_row][*current_column] == EMPTY;  // ...cell stays EMPTY in the next time step
                    } else {  // otherwise...
                        (*prob) = (*uniform)(*yarn);  // get random double between 0 and 1
                        if ((*prob) <= probSpread) {  // if prob is less than or equal to probSpread...
                            (*next_grid)[*current_row][*current_column] = YOUNG;  // ...cell becomes YOUNG in the next time step
                        } else {  // otherwise...
                            (*next_grid)[*current_row][*current_column] = EMPTY;  // ...cell stays EMPTY in the next time step
                        }
                    }
                    break;

                // if current cell is SPORE...
                case 1:
                    (*prob) = (*uniform)(*yarn);  // get random double between 0 and 1
                    if ((*prob) <= probSporeToYoung) {  // if prob is less than or equal to probSporeToYoung...
                        (*next_grid)[*current_row][*current_column] = YOUNG;  // ...cell becomes YOUNG in the next time step
                    } else {  // otherwise...
                        (*next_grid)[*current_row][*current_column] = SPORE;  // ...cell stays SPORE in the next time step
                    }
                    break;

                // if current cell is YOUNG...
                case 2:
                    (*next_grid)[*current_row][*current_column] = MATURING;  // ...cell becomes MATURING in the next time step
                    break;

                // if current cell is MATURING...
                case 3:
                    (*prob) = (*uniform)(*yarn);  // get random double between 0 and 1
                    if ((*prob) <= probMaturingToMushrooms) {  // if prob is less than or equal to probMaturingToMushrooms...
                        (*next_grid)[*current_row][*current_column] = MUSHROOMS;  // ...cell becomes MUSHROOMS in the next time step
                    } else {  // otherwise...
                        (*next_grid)[*current_row][*current_column] = OLDER;  // ...cell becomes OLDER in the next time step
                    }
                    break;

                // if current cell is MUSHROOMS...
                case 4:
                    (*next_grid)[*current_row][*current_column] = DECAYING;  // ...cell becomes DECAYING in the next time step
                    break;

                // if current cell is OLDER...
                case 5:
                    (*next_grid)[*current_row][*current_column] = DECAYING;  // ...cell becomes DECAYING in the next time step
                    break;

                // if current cell is DECAYING...
                case 6:
                    (*next_grid)[*current_row][*current_column] = DEAD;  // ...cell becomes DEAD in the next time step
                    break;

                // if current cell is DEAD...
                case 7:
                    (*next_grid)[*current_row][*current_column] = DEADER;  // ...cell becomes DEADER in the next time step
                    break;

                // if current cell is DEADER...
                case 8:
                    (*next_grid)[*current_row][*current_column] = DEPLETED;  // ...cell becomes DEPLETED in the next time step
                    break;

                // if current cell is DEPLETED...
                case 9:
                    (*prob) = (*uniform)(*yarn);  // get random double between 0 and 1
                    if ((*prob) <= probDepletedToSpore) {  // if prob is less than or equal to probDepletedToSpore...
                        (*next_grid)[*current_row][*current_column] = SPORE;  // ...cell becomes SPORE in the next time step
                    } else if ((*prob) <= probDepletedToEmpty) {  // if prob is less than or equal to probDepletedToEmpty...
                        (*next_grid)[*current_row][*current_column] = EMPTY;  // ...cell becomes EMPTY in the next time step
                    } else {  // otherwise...
                        (*next_grid)[*current_row][*current_column] = DEPLETED;  // ...cell stays DEPLETED in the next time step
                    }
                    break;

                // if current cell is INERT... (not currently used)
                case 10:
                        // note: there is the potential to initialize the grid with some cells starting out as inert
                            // representing spots where fungi cannot grow (rocks etc.) but this has not been implemented
                    (*next_grid)[*current_row][*current_column] = INERT;  // ...cell stays INERT in the next time step
                    break;
            }
        }
    }

    // copy next_grid onto current_grid
    copyGrid(current_grid, next_grid, ROWS, COLUMNS, current_row, current_column);
```

```

        // loop simulation for the next time step
    }
}

/* copyGrid() */
/* copies the contents of one grid into another grid of the same size */
void copyGrid(int ***current_grid, int ***next_grid, int * ROWS, int * COLUMNS, int * current_row, int * current_column) {
    for ((*current_row) = 1; (*current_row) <= (*ROWS); (*current_row)++) {  // for each row in the grid (except the ghost rows)...
        for ((*current_column) = 1; (*current_column) <= (*COLUMNS); (*current_column)++) {  // for each cell in that row...
            (*current_grid)[*current_row][*current_column] = (*next_grid)[*current_row][*current_column];  // ...store next_grid value in the same spot in current_grid
        }
    }
}

/* check_neighbors() */
/* checks the neighbors of a cell in the grid; returns 1 if at least one neighbor is YOUNG, otherwise returns 0 */
int check_neighbors(int ***current_grid, int * current_row, int * current_column, int * neighbor_row, int * neighbor_column) {
    for ((*neighbor_row) = (*current_row) - 1; (*neighbor_row) <= (*current_row) + 1; (*neighbor_row)++) {  // for each row in the 3x3 sub-grid...
        for ((*neighbor_column) = (*current_column) - 1; (*neighbor_column) <= (*current_column) + 1; (*neighbor_column)++) {  // for each cell in that row...
            if ( ((*neighbor_row) != (*current_row)) || ((*neighbor_column) != (*current_column)) ) {  // if that cell is a neighbor to the current cell...
                if ((*current_grid)[*neighbor_row][*neighbor_column] == YOUNG) {  // ... and if that neighbor is YOUNG...
                    return 1;  // return 1
                }
            }
        }
    }
    return 0;  // if none of the neighbors are YOUNG, return 0
}

/* deallocateGrid() */
/* deallocates the memory for the input grid */
void deallocateGrid(int ***grid, int * ROWS, int * current_row) {
    for ((*current_row) = 0; (*current_row) <= (*ROWS) + 1; (*current_row)++) {
        delete [] (*grid)[*current_row];
    }
    delete [] (*grid);
}

/* print_number_grid() */
/* prints the values in the input grid as numbers */
void print_number_grid(int ***grid, int * ROWS, int * COLUMNS, int * current_row, int * current_column) {
    for ((*current_row) = 0; (*current_row) <= (*ROWS) + 1; (*current_row)++) {  // for each row in the grid...

        // if current_row is the second row, add a row of dashes (to separate the ghost row)
        if ((*current_row) == 1) {
            for ((*current_column) = 0; (*current_column) <= (*COLUMNS) + 1; (*current_column)++) {
                printf("--");
            }
            // new line
            printf("\n");
        }

        for ((*current_column) = 0; (*current_column) <= (*COLUMNS) + 1; (*current_column)++) {  // for each cell in that row...

            // if current column is the second-from-the-left column, add a column of dashes (to separate the ghost column)
            if ((*current_column) == 1) { printf("| "); }

            // print value of current cell
            printf("%d ", (*grid)[*current_row][*current_column]);

            // if current column is the second-from-the-right columns, add a column of dashes (to separate the ghost column)
            if ((*current_column) == (*COLUMNS)) { printf("| "); }
        }

        // new line
        printf("\n");

        // if current row is the second-to-last row, add a row of dashes (to separate the ghost row)
        if ((*current_row) == (*ROWS)) {
            for ((*current_column) = 0; (*current_column) <= (*COLUMNS) + 1; (*current_column)++) {
                printf("--");
            }
            // new line
            printf("\n");
        }
    }
    // new line
    printf("\n");
}

/* print_colorful_grid() */
/* prints the values in the input grid as color-coded blocks */
void print_colorful_grid(int ***grid, int * ROWS, int * COLUMNS, int * current_row, int * current_column, int * current_value) {

    // print color key
    printf("\nKEY:\n-----------------------------------------\n");
    printf("|\tEMPTY\t\t|");
```

```
        black();
        printf("\t%lc\t", (wint_t)9608);
        reset_color();
    printf("|\n|\tSPORE\t\t|");
        red();
        printf("\t%lc\t", (wint_t)9547);
        reset_color();
    printf("|\n|\tYOUNG\t\t|");
        red();
        printf("\t%lc\t", (wint_t)9608);
        reset_color();
    printf("|\n|\tMATURING\t|");
        green();
        printf("\t%lc\t", (wint_t)9608);
        reset_color();
    printf("|\n|\tMUSHROOMS\t|");
        brown();
        printf("\t%lc\t", (wint_t)9608);
        reset_color();
    printf("|\n|\tOLDER\t\t|");
        brown();
        printf("\t%lc\t", (wint_t)9619);
        reset_color();
    printf("|\n|\tDECAYING\t|");
        purple();
        printf("\t%lc\t", (wint_t)9608);
        reset_color();
    printf("|\n|\tDEAD\t\t|");
        grey();
        printf("\t%lc\t", (wint_t)9619);
        reset_color();
    printf("|\n|\tDEADER\t\t|");
        grey();
        printf("\t%lc\t", (wint_t)9608);
        reset_color();
    printf("|\n|\tDEPLETED\t|");
        black();
        printf("\t%lc\t", (wint_t)9608);
        reset_color();
    // uncomment if using inert
    // printf("|\n|\tINERT\t\t|");
    //     black();
    //     printf("\t%lc\t", (wint_t)9608);
    //     reset_color();
    printf("|\n-----------------------------------------\n\n");


    for ((*current_row) = 0; (*current_row) <= (*ROWS) + 1; (*current_row)++) {  // for each row in the grid...

        // if current_row is the second row, add a row of dashes (to separate the ghost row)
        if ((*current_row) == 1) {
            for ((*current_column) = 0; (*current_column) <= (*COLUMNS) + 6; (*current_column)++) {
                printf("-");
            }
            // new line
            printf("\n");
        }

        for ((*current_column) = 0; (*current_column) <= (*COLUMNS) + 1; (*current_column)++) {  // for each cell in that row...

            // if current column is the second-from-the-left column, add a column of dashes (to separate the ghost column)
            if ((*current_column) == 1) { printf(" | "); }

            // get current cell's value
            (*current_value) = (*grid)[*current_row][*current_column];

            // print current cell's value as color symbol
            switch(*current_value) {

                // EMPTY
                case 0:
                    black();
                    printf("%lc", (wint_t)9608);
                    reset_color();
                    break;

                // SPORE
                case 1:
                    red();
                    printf("%lc", (wint_t)9547);
                    reset_color();
                    break;

                // YOUNG
                case 2:
                    red();
                    printf("%lc", (wint_t)9608);
                    reset_color();
                    break;

                // MATURING
```

```
                case 3:
                    green();
                    printf("%lc", (wint_t)9608);
                    reset_color();
                    break;

                // MUSHROOMS
                case 4:
                    brown();
                    printf("%lc", (wint_t)9608);
                    reset_color();
                    break;

                // OLDER
                case 5:
                    brown();
                    printf("%lc", (wint_t)9619);
                    reset_color();
                    break;

                // DECAYING
                case 6:
                    purple();
                    printf("%lc", (wint_t)9608);
                    reset_color();
                    break;

                // DEAD
                case 7:
                    grey();
                    printf("%lc", (wint_t)9619);
                    reset_color();
                    break;

                // DEADER
                case 8:
                    grey();
                    printf("%lc", (wint_t)9608);
                    reset_color();
                    break;

                // DEPLETED
                case 9:
                    black();
                    printf("%lc", (wint_t)9608);
                    reset_color();
                    break;

                // INERT (not currently used)
                case 10:
                    black();
                    printf("%lc", (wint_t)9608);
                    reset_color();
                    break;
            }

            // if current column is the second-from-the-right columns, add a column of dashes (to separate the ghost column)
            if ((*current_column) == (*COLUMNS)) { printf(" | "); }
        }

        // new line
        printf("\n");

        // if current row is the second-to-last row, add a row of dashes (to separate the ghost row)
        if ((*current_row) == (*ROWS)) {
            for ((*current_column) = 0; (*current_column) <= (*COLUMNS) + 6; (*current_column)++) {
                printf("-");
            }
            // new line
            printf("\n");
        }
    }
    // new line
    printf("\n");
}

/* reset_color() */
/* resets the text color for printf statements */
void reset_color() {
    printf("\033[0m");
}

/* black() */
/* sets the text color for printf statements to black */
void black() {
    printf("\033[0;30m");
}

/* red() */
/* sets the text color for printf statements to red */
void red() {
```

```
    printf("\033[0;31m");
}

/* green() */
/* sets the text color for printf statements to green */
void green() {
    printf("\033[0;32m");
}

/* brown() */
/* sets the text color for printf statements to brown */
void brown() {
    printf("\033[0;33m");
}

/* grey() */
/* sets the text color for printf statements to grey */
void grey() {
    printf("\033[1;34m");
}

/* purple() */
/* sets the text color for printf statements to purple */
void purple() {
    printf("\033[1;35m");
}

// end of file
```

----

fungi-omp.cpp:

```
/*******************************************************************************************
 * fungi-omp.cpp
 *******************************************************************************************
 *
 * simulates the growth of a mushroom network in a patch of grass in parallel using OpenMP
 *
 * created by Aron Smith-Donovan using code written by Libby Shoop as reference
 *
 * based on a project description posited in "Introduction to Computational Science:
 *      Modeling and Simulating for the Sciences" by Angela B. Shiflet and George W Shiflet
 *
 *
*/

/* LIBRARIES */
    #include <stdlib.h>
    #include <stdio.h>
    #include <string.h>
    #include <unistd.h>
    #include <cstdlib>
    #include <iostream>
    #include <trng/yarn2.hpp>
    #include <trng/uniform01_dist.hpp>
    #include <locale.h>
    #include <wchar.h>
    #include <omp.h>

/* UNIVERSAL CONSTANTS */
    // probability values for state changes
    #define probSpore 0.001             // probability that a site initially is SPORE
    #define probSporeToYoung 0.25       // probability that a SPORE will become YOUNG at the next time step
    #define probSpread 0.6              // probability that a EMPTY with a neighbor that is YOUNG will become YOUNG at the next time step
    #define probMaturingToMushrooms 0.7 // probability that a MATURING will become MUSHROOMS at the next time step (otherwise it becomes OLDER)
    #define probDepletedToSpore 0.0001  // probability that a DEPLETED will become SPORE at the next time step
    #define probDepletedToEmpty 0.5     // probability that a DEPLETED will become EMPTY at the next time step

    // cell states
    #define EMPTY 0      // empty ground containing no spore or hyphae
    #define SPORE 1      // contains at least one spore
    #define YOUNG 2      // young hyphae that cannot form mushrooms yet
    #define MATURING 3   // maturing hyphae that cannot form mushrooms yet
    #define MUSHROOMS 4  // older hyphae with mushrooms
    #define OLDER 5      // older hyphae with no mushrooms
    #define DECAYING 6   // decaying hyphae with exhausted nutrients
    #define DEAD 7       // newly dead hyphae with exhausted nutrients
    #define DEADER 8     // hyphae that have been dead for a while
    #define DEPLETED 9   // area whose nutrients have previously been depleted by fungal growth
    #define INERT 10     // inert area where plants cannot grow

/* FUNCTION DECLARATIONS */
void getArguments(int argc, char *argv[], int * ROWS, int * COLUMNS, int * TIME_STEPS, int * THREADS);
void allocateGrid(int ***grid, int * ROWS, int * COLUMNS);
void initializeGrid(int ***grid, int * ROWS, int * COLUMNS, double * prob, trng::yarn2 * yarn, trng::uniform01_dist<> * uniform);
```

```
void mushrooms(int ***current_grid, int ***next_grid, int * ROWS, int * COLUMNS, int * TIME_STEPS, int * current_value, double * prob, trng::yarn2 * yarn, trng::uniform01_dist<> * uniform);
void copyGrid(int ***current_grid, int ***next_grid, int * ROWS, int * COLUMNS);
int check_neighbors(int ***current_grid, int current_row, int current_column);
void deallocateGrid(int ***grid, int * ROWS);
void print_number_grid(int ***grid, int * ROWS, int * COLUMNS);
void print_colorful_grid(int ***grid, int * ROWS, int * COLUMNS, int * current_value);
void reset_color();
void black();
void red();
void green();
void brown();
void grey();
void purple();

/* main */
int main(int argc, char **argv){

    // declare shared variables
    double start_time, end_time, total_time;  // store timer values
    int ROWS, COLUMNS, TIME_STEPS, THREADS;  // store command line arguments
    int **current_grid;  // grid at current time step
    int **next_grid;  // grid at next time step
    // int current_row, current_column;  // grid cell counters
    // int current_time_step;  // time step counter
    // int neighbor_row, neighbor_column;  // check_neighbors() counters
    // int current_value;  // hold grid print values
    // double prob;  // stores randomly generated probability values



    // parse command line arguments
        // (need to do before parallel section to get the number of threads)
    getArguments(argc, argv, &ROWS, &COLUMNS, &TIME_STEPS, &THREADS);

    // start timing
    start_time = omp_get_wtime();

    // open parallel section
    // #pragma omp parallel
    // {

        // declare thread private variables
        int current_value;  // hold grid print values
        double prob;  // stores randomly generated probability values
        // int current_thread;  // stores thread rank

        // initialize RNG engine
        trng::yarn2 yarn;

        // seed RNG
        yarn.seed((long unsigned int)time(NULL));

        // split RNG by threads
        yarn.split(THREADS, omp_get_thread_num());

        // initialize RNG distribution function
        trng::uniform01_dist<> uniform;

        // allocate grids
        allocateGrid(&current_grid, &ROWS, &COLUMNS);
        allocateGrid(&next_grid, &ROWS, &COLUMNS);

        // initialize current_grid
        initializeGrid(&current_grid, &ROWS, &COLUMNS, &prob, &yarn, &uniform);

        // run the simulation
        mushrooms(&current_grid, &next_grid, &ROWS, &COLUMNS, &TIME_STEPS, &current_value, &prob, &yarn, &uniform);


    // }

    // end timing and print result
    end_time = omp_get_wtime();
    total_time = end_time - start_time;
    #ifdef DEBUG
        printf("\nruntime: %f seconds\n", total_time);
    #else
        printf("%f", total_time);
    #endif

    // deallocate grids
    deallocateGrid(&current_grid, &ROWS);
    deallocateGrid(&next_grid, &ROWS);

    // return statement
    return 0;

}

/* getArguments() */
```

```
/* fetches and stores command line arguments for # of rows, columns, time steps, and threads */
void getArguments(int argc, char *argv[], int * ROWS, int * COLUMNS, int * TIME_STEPS, int * THREADS) {

    // initialize variables
    int c;
    int rflag = 0;
    int cflag = 0;
    int sflag = 0;
    int tflag = 0;

    // retrieve command line arguments
    while ((c = getopt (argc, argv, "r:c:s:t:")) != -1) {
        switch (c) {
            case ’r’:
                rflag = 1;
                *ROWS = atoi(optarg);
                break;

            case ’c’:
                cflag = 1;
                *COLUMNS = atoi(optarg);
                break;

            case ’s’:
                sflag = 1;
                *TIME_STEPS = atoi(optarg);
                break;

            case ’t’:
                tflag = 1;
                *THREADS = atoi(optarg);
                omp_set_num_threads( atoi(optarg) );
                break;

            case ’?’:
                if (optopt == ’r’) {
                    fprintf (stderr, "Option -%c requires an argument.\n", optopt);
                } else if (optopt == ’c’) {
                    fprintf (stderr, "Option -%c requires an argument.\n", optopt);
                } else if (optopt == ’s’) {
                    fprintf (stderr, "Option -%c requires an argument.\n", optopt);
                } else if (optopt == ’t’) {
                    fprintf (stderr, "Option -%c requires an argument.\n", optopt);
                } else if (isprint (optopt)) {
                    fprintf (stderr, "Unknown option ‘-%c’.\n", optopt);
                } else {
                    fprintf (stderr, "Unknown option character ‘\\x%x’.\n", optopt);
                    exit(EXIT_FAILURE);
                }
        }
    }

    // check command line arguments
    if (rflag == 0) {
        fprintf(stderr, "Usage: %s -r number of rows\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (*ROWS < 1) {
        fprintf(stderr, "Usage: %s -r number of rows must be a positive nonzero integer\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (cflag == 0) {
        fprintf(stderr, "Usage: %s -c number of columns\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (*COLUMNS < 1) {
        fprintf(stderr, "Usage: %s -c number of columns must be a positive nonzero integer\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (sflag == 0) {
        fprintf(stderr, "Usage: %s -s number of time steps\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (*TIME_STEPS < 1) {
        fprintf(stderr, "Usage: %s -s number of time steps must be a positive nonzero integer\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (tflag == 0) {
        fprintf(stderr, "Usage: %s -t number of threads\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (*THREADS < 1) {
        fprintf(stderr, "Usage: %s -t number of threads must be a positive nonzero integer\n", argv[0]);
        exit(EXIT_FAILURE);
    }
}

/* allocateGrid() */
/* allocates enough space for the input grid to store the rows and columns for the problem */
void allocateGrid(int ***grid, int * ROWS, int * COLUMNS) {
    *grid = new int*[(*ROWS) + 2];    // create pointer array
```

```
    #pragma omp parallel for
    for (int current_row = 0; current_row <= (*ROWS) + 1; current_row++) {  // at each element in the pointer array...
        (*grid)[current_row] = new int[(*COLUMNS) + 2];  // ...create another pointer array
    }
}

/* initializeGrid() */
/* initializes the grid with empty spaces and spore spaces to begin the simulation */
void initializeGrid(int ***grid, int * ROWS, int * COLUMNS, double * prob, trng::yarn2 * yarn, trng::uniform01_dist<> * uniform) {
    #pragma omp parallel for collapse(2)
    for (int current_row = 1; current_row <= (*ROWS); current_row++) {  // for each row in the grid...
        for (int current_column = 1; current_column <= (*COLUMNS); current_column++) {  // for each cell in that row...
            (*prob) = (*uniform)(*yarn);  // get random double between 0 and 1
            if ((*prob) <= probSpore) {  // if prob is less than or equal to probSpore...
                (*grid)[current_row][current_column] = SPORE;  // ...then cell starts as SPORE
            } else {  // otherwise...
                (*grid)[current_row][current_column] = EMPTY;  // ...cell starts as EMPTY
            }
        }
    }

}

/* mushrooms() */
/* simulates the growth of mushroom networks into fairy rings */
void mushrooms(int ***current_grid, int ***next_grid, int * ROWS, int * COLUMNS, int * TIME_STEPS, int * current_value, double * prob, trng::yarn2 * yarn, trng::uniform01_dist<> * uniform) {
    for(int current_time_step = 0; current_time_step <= (*TIME_STEPS); current_time_step++) {  // for each time step... (note: time steps must happen sequentially)

        // set up ghost rows
        #pragma omp parallel for
        for (int ghost_column = 0; ghost_column <= (*COLUMNS) + 1; ghost_column++) {

            // set first row of grid to be the ghost of the second-to-last row
            (*current_grid)[0][ghost_column] = (*current_grid)[(*ROWS)][ghost_column];

            // set last row of grid to be the ghost of the second row
            (*current_grid)[(*ROWS) + 1][ghost_column] = (*current_grid)[1][ghost_column];
        }

        // set up ghost columns
        #pragma omp parallel for
        for (int ghost_row = 0; ghost_row <= (*ROWS) + 1; ghost_row++) {

            // set left-most column to be the ghost of the second-farthest-right column
            (*current_grid)[ghost_row][0] = (*current_grid)[ghost_row][*COLUMNS];

            // set right-most column to be the ghost of the second-farthest-left column
            (*current_grid)[ghost_row][(*COLUMNS) + 1] = (*current_grid)[ghost_row][1];
        }

        // DEBUG: display current grid
        #ifdef DEBUG
            #ifdef COLOR
                setlocale(LC_ALL, "");
                printf("\ntime step %d:\n", (current_time_step));
                print_colorful_grid(current_grid, ROWS, COLUMNS, current_value);
            #else
                printf("\ntime step %d:\n", (current_time_step));
                print_number_grid(current_grid, ROWS, COLUMNS);
            #endif
        #endif

        // determine grid at next time step
        #pragma omp parallel for collapse(2)
        for (int current_row = 1; current_row <= (*ROWS); current_row++) {  // for each row in the grid...
            for (int current_column = 1; current_column <= (*COLUMNS); current_column++) {  // for each cell in that row...

                (*current_value) = (*current_grid)[current_row][current_column];

                switch(*current_value) {

                    // if current cell is EMPTY...
                    case 0:
                        if (check_neighbors(current_grid, current_row, current_column) == 0) {  // if cell has no YOUNG neighbors...
                            (*next_grid)[current_row][current_column] == EMPTY;  // ...cell stays EMPTY in the next time step
                        } else {  // otherwise...
                            (*prob) = (*uniform)(*yarn);  // get random double between 0 and 1
                            if ((*prob) <= probSpread) {  // if prob is less than or equal to probSpread...
                                (*next_grid)[current_row][current_column] = YOUNG;  // ...cell becomes YOUNG in the next time step
                            } else {  // otherwise...
                                (*next_grid)[current_row][current_column] = EMPTY;  // ...cell stays EMPTY in the next time step
                            }
                        }
                        break;

                    // if current cell is SPORE...
                    case 1:
                        (*prob) = (*uniform)(*yarn);  // get random double between 0 and 1
                        if ((*prob) <= probSporeToYoung) {  // if prob is less than or equal to probSporeToYoung...
```

```
                            (*next_grid)[current_row][current_column] = YOUNG;  // ...cell becomes YOUNG in the next time step
                        } else {  // otherwise...
                            (*next_grid)[current_row][current_column] = SPORE;  // ...cell stays SPORE in the next time step
                        }
                        break;

                    // if current cell is YOUNG...
                    case 2:
                        (*next_grid)[current_row][current_column] = MATURING;  // ...cell becomes MATURING in the next time step
                        break;

                    // if current cell is MATURING...
                    case 3:
                        (*prob) = (*uniform)(*yarn);  // get random double between 0 and 1
                        if ((*prob) <= probMaturingToMushrooms) {  // if prob is less than or equal to probMaturingToMushrooms...
                            (*next_grid)[current_row][current_column] = MUSHROOMS;  // ...cell becomes MUSHROOMS in the next time step
                        } else {  // otherwise...
                            (*next_grid)[current_row][current_column] = OLDER;  // ...cell becomes OLDER in the next time step
                        }
                        break;

                    // if current cell is MUSHROOMS...
                    case 4:
                        (*next_grid)[current_row][current_column] = DECAYING;  // ...cell becomes DECAYING in the next time step
                        break;

                    // if current cell is OLDER...
                    case 5:
                        (*next_grid)[current_row][current_column] = DECAYING;  // ...cell becomes DECAYING in the next time step
                        break;

                    // if current cell is DECAYING...
                    case 6:
                        (*next_grid)[current_row][current_column] = DEAD;  // ...cell becomes DEAD in the next time step
                        break;

                    // if current cell is DEAD...
                    case 7:
                        (*next_grid)[current_row][current_column] = DEADER;  // ...cell becomes DEADER in the next time step
                        break;

                    // if current cell is DEADER...
                    case 8:
                        (*next_grid)[current_row][current_column] = DEPLETED;  // ...cell becomes DEPLETED in the next time step
                        break;

                    // if current cell is DEPLETED...
                    case 9:
                        (*prob) = (*uniform)(*yarn);  // get random double between 0 and 1
                        if ((*prob) <= probDepletedToSpore) {  // if prob is less than or equal to probDepletedToSpore...
                            (*next_grid)[current_row][current_column] = SPORE;  // ...cell becomes SPORE in the next time step
                        } else if ((*prob) <= probDepletedToEmpty) {  // if prob is less than or equal to probDepletedToEmpty...
                            (*next_grid)[current_row][current_column] = EMPTY;  // ...cell becomes EMPTY in the next time step
                        } else {  // otherwise...
                            (*next_grid)[current_row][current_column] = DEPLETED;  // ...cell stays DEPLETED in the next time step
                        }
                        break;

                    // if current cell is INERT... (not currently used)
                    case 10:
                            // note: there is the potential to initialize the grid with some cells starting out as inert
                                // representing spots where fungi cannot grow (rocks etc.) but this has not been implemented
                        (*next_grid)[current_row][current_column] = INERT;  // ...cell stays INERT in the next time step
                        break;
                }
            }
        }

        // copy next_grid onto current_grid
        copyGrid(current_grid, next_grid, ROWS, COLUMNS);

        // loop simulation for the next time step
    }
}

/* copyGrid() */
/* copies the contents of one grid into another grid of the same size */
void copyGrid(int ***current_grid, int ***next_grid, int * ROWS, int * COLUMNS) {
    #pragma omp parallel for collapse(2)
    for (int current_row = 1; current_row <= (*ROWS); current_row++) {  // for each row in the grid (except the ghost rows)...
        for (int current_column = 1; current_column <= (*COLUMNS); current_column++) {  // for each cell in that row...
            (*current_grid)[current_row][current_column] = (*next_grid)[current_row][current_column];  // ...store next_grid value in the same spot in current_grid
        }
    }
}

/* check_neighbors() */
/* checks the neighbors of a cell in the grid; returns 1 if at least one neighbor is YOUNG, otherwise returns 0 */
int check_neighbors(int ***current_grid, int current_row, int current_column) {
    int young = 0;  // young counter
```

```
    #pragma omp parallel for collapse(2) reduction(+:young)
    for (int neighbor_row = current_row - 1; neighbor_row <= current_row + 1; neighbor_row++) {  // for each row in the 3x3 sub-grid...
        for (int neighbor_column = current_column - 1; neighbor_column <= current_column + 1; neighbor_column++) {  // for each cell in that row...
            if ( (neighbor_row != current_row) || (neighbor_column != current_column) ) {  // if that cell is a neighbor to the current cell...
                if ((*current_grid)[neighbor_row][neighbor_column] == YOUNG) {  // ... and if that neighbor is YOUNG...
                    young += 1;  // ... increase young counter by 1
                }
            }
        }
    }
    if (young == 0) {  // if none of the neighbors are YOUNG...
        return 0;  // ...return 0
    } else {  // otherwise...
        return 1;  // return 1
    }

}

/* deallocateGrid() */
/* deallocates the memory for the input grid */
void deallocateGrid(int ***grid, int * ROWS) {
    #pragma omp parallel for
    for (int current_row = 0; current_row <= (*ROWS) + 1; current_row++) {
        delete [] (*grid)[current_row];
    }
    delete [] (*grid);
}

/* print_number_grid() */
/* prints the values in the input grid as numbers */
void print_number_grid(int ***grid, int * ROWS, int * COLUMNS) {
    for (int current_row = 0; current_row <= (*ROWS) + 1; current_row++) {  // for each row in the grid...

        // if current_row is the second row, add a row of dashes (to separate the ghost row)
        if (current_row == 1) {
            for (int i = 0; i <= (*COLUMNS) + 1; i++) {
                printf("--");
            }
            // new line
            printf("\n");
        }

        for (int current_column = 0; current_column <= (*COLUMNS) + 1; current_column++) {  // for each cell in that row...

            // if current column is the second-from-the-left column, add a column of dashes (to separate the ghost column)
            if (current_column == 1) { printf("| "); }

            // print value of current cell
            printf("%d ", (*grid)[current_row][current_column]);

            // if current column is the second-from-the-right columns, add a column of dashes (to separate the ghost column)
            if (current_column == (*COLUMNS)) { printf("| "); }
        }

        // new line
        printf("\n");

        // if current row is the second-to-last row, add a row of dashes (to separate the ghost row)
        if (current_row == (*ROWS)) {
            for (int j = 0; j <= (*COLUMNS) + 1; j++) {
                printf("--");
            }
            // new line
            printf("\n");
        }
    }
    // new line
    printf("\n");
}

/* print_colorful_grid() */
/* prints the values in the input grid as color-coded blocks */
void print_colorful_grid(int ***grid, int * ROWS, int * COLUMNS, int * current_value) {

    // print color key
    printf("\nKEY:\n-----------------------------------------\n");
    printf("|\tEMPTY\t\t|");
        black();
        printf("\t%lc\t", (wint_t)9608);
        reset_color();
    printf("|\n|\tSPORE\t\t|");
        red();
        printf("\t%lc\t", (wint_t)9547);
        reset_color();
    printf("|\n|\tYOUNG\t\t|");
        red();
        printf("\t%lc\t", (wint_t)9608);
        reset_color();
    printf("|\n|\tMATURING\t|");
        green();
        printf("\t%lc\t", (wint_t)9608);
```

```
        reset_color();
    printf("|\n|\tMUSHROOMS\t|");
        brown();
        printf("\t%lc\t", (wint_t)9608);
        reset_color();
    printf("|\n|\tOLDER\t\t|");
        brown();
        printf("\t%lc\t", (wint_t)9619);
        reset_color();
    printf("|\n|\tDECAYING\t|");
        purple();
        printf("\t%lc\t", (wint_t)9608);
        reset_color();
    printf("|\n|\tDEAD\t\t|");
        grey();
        printf("\t%lc\t", (wint_t)9619);
        reset_color();
    printf("|\n|\tDEADER\t\t|");
        grey();
        printf("\t%lc\t", (wint_t)9608);
        reset_color();
    printf("|\n|\tDEPLETED\t|");
        black();
        printf("\t%lc\t", (wint_t)9608);
        reset_color();
    // uncomment if using inert
    // printf("|\n|\tINERT\t\t|");
    //     black();
    //     printf("\t%lc\t", (wint_t)9608);
    //     reset_color();
    printf("|\n-----------------------------------------\n\n");

    for (int current_row = 0; current_row <= (*ROWS) + 1; current_row++) {  // for each row in the grid...

        // if current_row is the second row, add a row of dashes (to separate the ghost row)
        if (current_row == 1) {
            for (int i = 0; i <= (*COLUMNS) + 6; i++) {
                printf("-");
            }
            // new line
            printf("\n");
        }

        for (int current_column = 0; current_column <= (*COLUMNS) + 1; current_column++) {  // for each cell in that row...

            // if current column is the second-from-the-left column, add a column of dashes (to separate the ghost column)
            if (current_column == 1) { printf(" | "); }

            // get current cell's value
            (*current_value) = (*grid)[current_row][current_column];

            // print current cell's value as color symbol
            switch(*current_value) {

                // EMPTY
                case 0:
                    black();
                    printf("%lc", (wint_t)9608);
                    reset_color();
                    break;

                // SPORE
                case 1:
                    red();
                    printf("%lc", (wint_t)9547);
                    reset_color();
                    break;

                // YOUNG
                case 2:
                    red();
                    printf("%lc", (wint_t)9608);
                    reset_color();
                    break;

                // MATURING
                case 3:
                    green();
                    printf("%lc", (wint_t)9608);
                    reset_color();
                    break;

                // MUSHROOMS
                case 4:
                    brown();
                    printf("%lc", (wint_t)9608);
                    reset_color();
                    break;

                // OLDER
                case 5:
```

```
                    brown();
                    printf("%lc", (wint_t)9619);
                    reset_color();
                    break;

                // DECAYING
                case 6:
                    purple();
                    printf("%lc", (wint_t)9608);
                    reset_color();
                    break;

                // DEAD
                case 7:
                    grey();
                    printf("%lc", (wint_t)9619);
                    reset_color();
                    break;

                // DEADER
                case 8:
                    grey();
                    printf("%lc", (wint_t)9608);
                    reset_color();
                    break;

                // DEPLETED
                case 9:
                    black();
                    printf("%lc", (wint_t)9608);
                    reset_color();
                    break;

                // INERT (not currently used)
                case 10:
                    black();
                    printf("%lc", (wint_t)9608);
                    reset_color();
                    break;
            }

            // if current column is the second-from-the-right columns, add a column of dashes (to separate the ghost column)
            if (current_column == (*COLUMNS)) { printf(" | "); }
        }

        // new line
        printf("\n");

        // if current row is the second-to-last row, add a row of dashes (to separate the ghost row)
        if (current_row == (*ROWS)) {
            for (int j = 0; j <= (*COLUMNS) + 6; j++) {
                printf("-");
            }
            // new line
            printf("\n");
        }
    }
    // new line
    printf("\n");
}

/* reset_color() */
/* resets the text color for printf statements */
void reset_color() {
    printf("\033[0m");
}

/* black() */
/* sets the text color for printf statements to black */
void black() {
    printf("\033[0;30m");
}

/* red() */
/* sets the text color for printf statements to red */
void red() {
    printf("\033[0;31m");
}

/* green() */
/* sets the text color for printf statements to green */
void green() {
    printf("\033[0;32m");
}

/* brown() */
/* sets the text color for printf statements to brown */
void brown() {
    printf("\033[0;33m");
}

```

```c
/* grey() */
/* sets the text color for printf statements to grey */
void grey() {
    printf("\033[1;34m");
}

/* purple() */
/* sets the text color for printf statements to purple */
void purple() {
    printf("\033[1;35m");
}

// end of file
```