CSE 6010
Assignment 5
Reaction-Diffusion System with OpenMP

Initial Submission Due Date: 11:59pm on Thursday, November 14
Final Submission Due Date: 11:59pm on Thursday, November 21
(No peer review assignment)
Submit Makefile and code as described herein to Gradescope through Canvas
48-hour grace period applies to all deadlines

In this assignment, you will implement a solution for a reaction-diffusion system in C using OpenMP to improve performance. Reaction-diffusion systems are coupled in space and evolve in time according to differential equations (which you will not need to worry about for this assignment). Such systems can exhibit interesting patterns and other phenomena. Here we will be concerned with implementing a flexible system that can accommodate spatial domains in either one or two dimensions (as represented by a "1D" or "2D" array) and that can support two approaches for handling diffusion at the boundaries of the domain. The "reaction" component will be calculated using a function specified in a provided file, so that you will only need to calculate the diffusion term. Each location in the domain will be updated iteratively.

Note that this assignment will have some commonality with Assignment 2 in terms of handling a two-dimensional domain, but more complexity will be added as well as the use of OpenMP.

Specifications:

- Input data for the system will be provided through the file provided as the first command-line argument; examples will be provided. This file will include the following information in order, delimited by spaces:
    - DIMS: the number of spatial dimensions, either 1 or 2
    - SIZE: for a 1-dimensional system, the number of grid points; for a 2-dimensional system, the number of grid points along each side (so that the total number of grid points is SIZE * SIZE)
    - NUMITERATIONS: the number of iterations over which to evolve the system
    - REGIONEXTENT: the number of grid points in each dimension that will be set to a specified initial value (see below)
    - INITIALVALUE: the initial value to use in the specified region
    - PERIODIC: set to 1 for periodic ("wrap-around") conditions along all boundaries or to 0 for reflecting conditions along all boundaries
    - Any additional values should be ignored (i.e., you do not need to read any further)

  An example file will be provided for your use, but you should test variations, as the assignment will be graded with different input files to test for correctness under a variety of conditions.
- Command-line arguments: Your code should take exactly two required command-line arguments, first, the name of the file containing the input data, and second, the number of threads to use in the parallel region.

- Data structures: the file “model.h” defines the Node data structure that will be used in this assignment. The Node struct has 13 members. Functions to initialize a Node struct at a single grid point and to update the Node struct for the reaction term (combined with a parameter that passes the diffusion term you will write code to calculate) at a single grid point are specified in “model.c.” You should call these functions appropriately from your `main()` function.
- You should use dynamic memory allocation to create the grid as a one-dimensional or “two-dimensional” array, depending on the values of DIMS. For the two-dimensional case, you should store the grid as a one-dimensional array: store the value at two-dimensional grid location (i,j) at one-dimensional array index `i*SIZE+j`, as you have done in previous assignments.
- Initializing your array of nodes: Initialization consists of two parts.
    - First, you should call the `initializeNode()` function (declared in model.h and defined in model.c) to initialize each node. This function takes a pointer to a Node as an argument and sets all members of that struct to particular values.
    - Second, for a subset of nodes, you should set just the member named “v” to the value INITIALVALUE defined in reactiondiffusion.h. The size of the subset will be given by REGIONEXTENT also defined in reactiondiffusion.h.
        - For the one-dimensional case, after initializing all nodes in the domain, you should further initialize just the “v” member to INITIALVALUE for the first REGIONEXTENT nodes: that is, for the nodes at indices 0 to REGIONEXTENT-1 within the array.
        - For the two-dimensional case, after initializing all nodes in the domain, you should further initialize just the “v” member to INITIALVALUE for the first REGIONEXTENT nodes in each direction: that is, for the nodes at indices 0 to REGIONEXTENT-1 in both spatial dimensions within the array. You should apply the setting to REGIONEXTENT* REGIONEXTENT total nodes.
- Calculating diffusion: In this case we will be concerned only with diffusion in the “v” member of the node struct. The diffusion term at a node with index `i` along a single direction is proportional to the sum of the “v” member at the two neighboring nodes minus twice the value of the “v” member at the node itself. The proportionality constant will be handled by the calculations in model.c, so you will just need to perform this sum. Of course, there are a few complexities concerning what to do at the boundaries and how to extend to two dimensions.
    - Boundaries: Your code should be able to handle two types of boundaries according to the setting in input file.
        - Reflecting boundaries: If you are calculating the diffusion value for a node located at the edge of the grid (e.g., in one dimension this would include node 0 and node SIZE-1), you should obtain the needed value that would be off-grid by “reflecting” back in the opposite direction. For example, node 0’s right neighbor will be node 1, but its left neighbor will also be node 1. Similarly, node SIZE-1’s right neighbor will be the same as its left neighbor, node SIZE-2.
        - Periodic boundaries: For the same situation above, you will instead obtain the needed value that would be off-grid by “wrapping around”.

For example, node 0's right neighbor will be node 1, but its left neighbor will be node SIZE-1. Similarly, node SIZE-1's right neighbor will be node 0.

  - Dimensions: In two dimensions, your code will need to calculate a second diffusion term in the other direction ("up-down"), along with the "left-right" direction. This diffusion term should be added to the term calculated in the "left-right" direction but otherwise will be calculated the same way, including handling boundaries. All boundaries should be handled according to the method specified in the input file. For example, if PERIODIC is set to 1, then both the left-right and up-down boundaries should be handled periodically; in this case, the node with two-dimensional indices [2][0] would have neighbors in the left-right direction of [1][0] and [3][0], and in the up-down direction the neighbors would have indices of [2][SIZE-1] and [2][1].
  - Note that if you are careful, you can write your code such that it works for the one-dimensional and two-dimensional cases without redundancy and special conditions, except for calculating diffusion in the extra dimension for the two-dimensional case. Lengthy conditionals will not be needed.
- Iterations: In each iteration your code will need to perform the following.
  - Calculate the diffusion value for each node according to the number of dimensions and boundary condition type specified. When calculating the diffusion value, you must use the same "old" values for all nodes.
  - Calculate each updated node value by calling `updateNode()`, passing in the diffusion value for that node as the third parameter of the function. This function will take the values of the Node that is the first parameter of the function and use them to calculate the updated values that will be stored in the Node that is the second parameter of the function. (Actually the parameters are pointers to the Nodes.)
  - Copy the updated node values back to the original nodes so that they will be available for the following iteration.
- Parallelization:
  - Your code should implement parallelization at one or more appropriate places to achieve a performance improvement with multiple threads.
  - The number of threads should be set to the value given on the command-line.
  - Immediately before and immediately after the parallel region(s) in your code (for example, including the main iterations), call the function `omp_get_wtime()` to obtain the times and accumulate the elapsed time within the parallel region (as in Workshop 5), so that you can later output the total time spent within the parallel regions.
  - In addition to correctness, your code will be tested to ensure speedup is achieved with more threads under suitable conditions. The relative complexity of the calculations performed in `updateNode()` will allow you to see speedup for modest problem sizes.
  - As in Workshop 5, depending on your operating system and other system properties, you may not see any speedup with multiple threads. If this is the case for you, please try the autograder environment, which should demonstrate speedup.

- Output:
    - Screen output: At the end of the program, write out the cumulative time your code spent in the parallel region (see above). Write this single value followed by a newline character using the %f format specifier.
    - File output: At the end of the program, write out the final array values to a file named “output.txt”. Write out each value in order followed by a newline character. For two-dimensional cases, output the values as they are stored in memory (one row at a time), with one value at a time, for a total of SIZE*SIZE lines.

To receive full credit, your code must be well structured and documented so that it is easy to understand. Be sure to include comments that explain your code statements and structure. Your code should avoid redundancy and overly complex solutions, and it should achieve speedup for suitably large cases. In addition, you should use #include to include model.h, and you should compile the code you write together with the file model.c.

Initial submission (worth 1 point): Submit your code to the Initial Submission on Gradescope. Your initial submission will not be graded for correctness or even tested but rather will be graded based on the appearance of a good-faith effort to complete the majority of the assignment. Any time you wish to test your code for correctness, please submit it to the Final Submission on Gradescope.

Final submission (worth 11 points): Submit your code to the Final Submission on Gradescope. You may submit any number of files named in any way you like, but you will be required to submit a Makefile that will be used to compile your program, and the name of your executable must be **reactiondiffusion** (no extension). Do not zip the files or upload a directory; instead, submit your files directly. **Only submit the source code files that you wrote yourself; do not include the executable or provided files. Do not modify or submit the model.c and model.h files.**

Sample input and output: A sample input file with its matching sample output file are provided on Canvas.

Some hints:

- Start early!
- Identify a logical sequence for implementing the desired functionality. Then implement one piece at a time and verify each piece works properly before proceeding to implement the next. Working through the tests outlined below in sequence may be helpful.
- Complete a serial version of the code before adding parallelization (but plan ahead!).
- Be sure to test for speedup using the autograder.
- Debugging reaction-diffusion problems is challenging, as errors are not always obvious from observing the output. Coming to office hours to discuss your implementation and walk through the logic of your program will be the best way to resolve this type of issue, so please plan ahead to have enough time to debug the assignment before the deadline.
- For this assignment, there is no need to attempt to parallelize file I/O.

Grading

Program correctness will be assessed using the autograder on Gradescope; valgrind will be used to detect memory errors, including failure to free all allocated memory. Detailed grading specifications are described below. Numbers in parentheses indicate the total points available for a specific component.

1. **Initial submission (1)**: Graded manually. The point is earned for making meaningful progress toward completing the assignment. The program does not need to compile or run correctly at this stage.
2. **Autograder (8)**
    a. **Test 0: Compile the program (0)**: This test is not worth any points, but it will show the output if errors are encountered compiling your program on the autograder. This output should be helpful if your code compiles on your machine but not on the autograder. Fix these errors before moving on to anything else!
    b. **Test 1: Correctness (6):** Match the correct output file for six different input files testing different parameters. Each correct output is worth one point. The output must be correct regardless of the number of threads used.
    c. **Test 2: Speedup (2):** Achieve a speedup factor of 2 or greater when using 4 threads. Two cases will be tested using a relatively large domain size and iteration count. Each case is worth one point.
3. **Program structure and implementation (1.5)**: Graded manually. Points are earned for implementing the algorithms and data structures correctly as described in the assignment and making reasonable decisions when designing your program.
4. **Program style and documentation (1.5)**: Graded manually. Points are earned for using abstractions to avoid redundant code and making your program readable by using appropriate comments, variable names, whitespace, indentation, etc. Write your code so someone else can easily follow what is going on.

The autograder will run immediately when the program is submitted and should present the results quickly. The manually graded components of the assignment will not be available until after the grades are published. Autograder feedback will only be available with the final submission for the assignment, as the initial submission will not be graded for correctness.

Note that autograder points may be deducted if the instructions of the assignment are not followed, for example, if the program does not set the number of threads to the value of the command-line argument. Additionally, autograder points will be manually deducted if the program output is manually set to pass the autograder without correctly implementing the solution.