CX 4010 / CSE 6010
Assignment 5
Diffusion
Additional requirements for graduate students only

Initial Submission Due Date: 11:59pm on **Thursday, November 17**
Final Submission Due Date: 11:59pm on **Tuesday, November 29**
Submit codes as a single zipfile as described herein to Canvas
48-hour grace period applies to all deadlines

In this assignment, you will write a program to simulate diffusion of heat along a one-dimensional representation of a rod using OpenMP. Essentially, you will use an array to represent the temperature at evenly spaced locations along the rod. (You will not really need to consider the physical explanation of the problem in order to write the code, but you may find it helpful.)

Diffusion calculation:

You will solve a simplified heat diffusion equation in one dimension, which amounts to working with a one-dimensional array.

Initialization: The size of the array N should be set dynamically using malloc based on the command-line argument (see below). (Don't forget to free any memory allocated dynamically!) The temperature for all N locations should be set to **0** initially except for one specific location corresponding to a single array element that is heated initially. Undergraduate students will set the value at that location to **50**; graduate students will vary the value (see below) but will set it initially to **50**. The location of the initially heated element either is given by a command-line argument (if present) or is set as a random value along the array representing the rod. The random value should be chosen using a time-dependent seed, so that the location will be different every time the program is run.

Solution: Heat slowly diffuses along the rod. Although we will not use physical units, the same idea applies, so time will be represented by updating the temperature repeatedly through iteration. Your program should perform **2000** iterations. In each iteration, the temperature should be updated at each location as follows:

$$heatnew[i] = heat[i] + scale * \left(heat_{left,i} + heat_{right,i} - 2 * heat[i]\right).$$

Here $heat_{left}$ and $heat_{right}$ represent the left and right neighboring elements along the rod (array), respectively. In the interior of the array, $heat_{left,i}$ for location i would correspond to $heat[i-1]$; similarly, $heat_{right,i}$ would correspond to $heat[i+1]$. At the left edge ($i == 0$), use $heat[1]$ as both the left and right neighbors, and at the right edge ($i == N-1$), use $heat[N-2]$ as both the left and right neighbors. (These choices implement so-called "no-flux" boundaries.) You should use a fixed value of $scale = \mathbf{0.1}$.

Note that you will need to have a second array to hold updated values; otherwise, the update equations for different array indices will be solved using a mix of old and new values. At the end of a single iteration you will need to ensure that the updated values are available to use on the right-hand side of the equation for the next iteration. There are a couple of ways to do this; it is fine if you simply copy the values of $heatnew$ into the corresponding locations in $heat$.

You should write a function to calculate this solution. Use OpenMP to parallelize the computations by assigning the update equation above to different threads for the different array elements. You may assign work to the threads in any reasonable way you like. Consider where barriers may be necessary.

At the end of 2000 iterations, your function should return a single double-precision value, which is the maximum value of the array at that time. Undergraduate students should print this return value from the main function.

Modified binary search (graduate students only):

Graduate students will be required to use a modified binary search to find the maximum value of the initially heated location such that the maximum value in the array after **2000** iterations is less than **1**. You can treat the value of the initially heated location as an integer and just search to find the largest integer that achieves the condition of the maximum value in the array being less than 1. The idea will be to use a binary-search-type approach to bracket the desired value and then progressively narrow down the range of possible values by half (roughly) with each iteration of the search.

For the modified binary search, you should start by setting the value of the initially heated location (H_{init}) to **50**. If the return value of the function is more than 1, then set $L = 0$ and $H = 50$, and set $H_{init} = (L + H)/2$, the midpoint of the bracketed region. Evaluate the function again using this value of H_{init} to determine whether the return value is less than 1 at the midpoint of the bracketed region. Adjust the values for H and L in a manner analogous to binary search. Continue until you find the largest integer value for H_{init} that will keep the return value of the function below 1 (you should work out an appropriate stopping condition).

If the return value of the function using $H_{init} = 50$ is not more than 1, double H_{init} until you find the return value is above 1. At this point, H_{init} gives a return value that is more than 1 but the previous value, currently $H_{init}/2$, gives a value that is not more than 1. Thus, you should set $H = H_{init}$ and $L = H_{init}/2$ and proceed as described above, using this bracketed region.

At the end, your main program should print out both the value of H_{init} found as well as the function return value using that value for H_{init}.

Command-line arguments:

The first command-line argument, which is required, is the number of elements in the rod (size of the one-dimensional array representing the temperature). If no command-line arguments are specified, there should be an error, but you may assume that any command-line argument for the array size will be reasonable; you do not need to perform any validation.

The second command-line argument, which is optional, is used to specify the location of the initially heated element within the one-dimensional array. You may assume that the value is specified properly; you are not required to perform any validation. If the second command-line argument is not used, the program should choose a random index along the array representing the rod as the location of the initially heated element.

The third command-line argument, which is optional, is used to specify the number of threads OpenMP should use. Use of the third command-line argument requires use of the first and second command-line arguments. You are not required to perform any validation for this value.

Apart from the requirements indicated above, you may organize your program any way you like, including the use of any supporting .h and .c files. Your main program should be named **diffusion.c**.

Your code (final submission) should successfully compile (using gcc) and run on the COC-ICE cluster. This does not mean we necessarily will try to run your code on the cluster, but if we have any difficulties compiling and running it, we will move it to that environment for further testing. If you are compiling and running your codes under linux, most likely you will not need to make any changes as a result of this requirement.

OMP performance: After your program is fully developed, you should time the program for several different numbers of threads and compare the runtimes; you will comment on performance in your slides. If you use linux, you are welcome to use the time command; just type "time ./myprogram" instead of "./myprogram" and use the "real" time (first value returned) as the runtime in seconds. Include comments on OMP performance in your submission slides.

OMP performance note: To keep things simple for your first OpenMP program, the calculations are not very complex. For this reason, it is quite likely you will not see a speedup using OpenMP as you continually increase the number of threads. In fact, you may even see the performance degrade. The reason is that there may not be enough work being done by each thread to offset (1) the overhead involved in thread creation and deletion and (2) the memory costs associated with accessing the array entries and performing the calculations.

Special OMP note for Mac users: Beyond the above discussion, for reasons we are not going to get into, it is highly likely that you will not observe a speedup (or any difference in time at all) using OpenMP regardless of how many threads you use. We strongly recommend using the COC-ICE cluster; you should see a time difference there. It should be ok to run on the login nodes as long as your program runs for less than a minute or so.

Reference value: As an example, you should find that for an array size of 6000 and an initial location of 3000, after 2000 iterations the maximum value is 0.9975. However, your code will be tested using different sizes and locations.

You should submit to Canvas a single zipfile that is named according to your Georgia Tech login—the part that precedes @gatech.edu in your GT email address. 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.

The zipfile should include the following files:

(1) your code (all .c and .h files). If you are using linux or Mac OS, you may use a makefile to compile and run your program, and you should include it if so. Do not include any input or output files.

(2) a README text file (not formatted in a word processor, for example) that includes the compiler and operating system you used for compiling and running your code along with instructions on how to compile and run your program.

(3) a series of 3 slides composed in PowerPoint or similar software, saved either in PowerPoint or as a PDF and named slides.pptx or slides.pdf, structured as follows:

- Slide 1: your name and a brief explanation of how you developed/structured your program (data structures, functions, program flow, etc.). You are limited to one slide.
- Slide 2: a brief statement explaining why you believe your program works correctly. You are again limited to one slide; focus on what you think is most important.
- Slide 3: your assessment of OMP performance for this problem. Include a table of runtimes using several numbers of threads for one or more problem sizes (array sizes) and your comments on what you observed.

Initial submission

Your initial submission does not need to include the slides. It will not be graded for correctness or even tested (including for compatibility with gcc on the COC-ICE cluster) but rather will be graded based on the appearance of a good-faith effort to complete the majority of the assignment. **For the initial submission, it is fine to write a serial implementation (without OpenMP).**