Assignment 3
Auto-generating a maze
Due date: Friday, November 19 11:59 pm

Description

Implement that depth first search algorithm that creates a path through a two dimensional array. This path would represent a 2D maze. Once it is running in serial add the OpenMP commands to make it work in parallel with four processors.

You do not need to run any timing tests for this assignment.

The program starts with a full array with no unclaimed space. As the algorithm runs it will extend the path it creates as it fills the array. When the algorithm ends it should have filled the array. The paths that are created by the algorithm must be separated by walls so there cannot be two paths adjacent to each other.

Maze Creation Algorithm

1. Pick a starting location and put it in the stack. Fill the starting location with the thread rank.
2. While the stack is not empty.
3. Pop an item out of the stack.
4. Randomly visit each of the four neighbours. If the neighbour has not been claimed already (it contains a `.`) then claim it and the intervening unclaimed space by putting the thread rank in those two array elements. If the neighbour contains anything other than a `.` then leave it alone.
5. Push each of the successfully visited neighbours onto the stack. Push only the neighbours that are two spaces from the starting point.

Visiting a neighbour involves changing the filled values of the array `.` to values that represent the thread. Each thread will replace the `.` with the thread rank from 0 to 3. The neighbour visited is two spaces away from the current location. This means that when the neighbour is claimed by the thread that it will actually change two values in the array.

In the following example, thread 1 is testing if it can claim the space to the right. It tests the space that is two elements to the right.

```
...
1..
...
```

The space has not been claimed so it will mark both spaces with its rank to add them to the maze. It would look like this:

```
...
111
...
```

If instead it found that the space two elements away contained something other than a period then it could not add those elements to the maze. In the following example, thread 1 has already claimed the space that is two characters to the right so it cannot be linked to the current location. The same would be true if the space had been claimed by any of the other threads and contained numbers 0, 2, or 3.

```
...
1.1
...
```

Do not consider diagonals as neighbours.

Randomly visiting the neighbours means that they should not always be visited in the same order. Use a random number generator to pick the order each time neighbours are visited.

Data Structures

Allocate space for and initialize the maze array. The initial values in the array are periods `.` . These will be replaced as the maze is created - each thread will replace the `.` with the thread id.

Create the stack data structures for each thread. The stack stores locations that were previously visited. You can implement the stack using either an array or a list. You are also welcome to reuse the stack from a previous course, as long as the instructor from that course allows it.

The program will use one maze array. Since this is shared by all stacks, access to it is a critical section.

Each thread will use its own stack, so updates to these wont be in a critical section.

Implementation Notes

You will need to submit code that compiles into two executables. The serial executable should be named `maze` and the parallel version named `mazep`. Submit one `maze.c` file and a makefile that creates both executable from the single `.c` file.

Unclaimed space is the maze is marked with a `.` and claimed space is marked with numbers `0`, `1`, `2`, `3`. No other characters should appear in the maze.

The outer rows and columns of the array are a border around maze. Rows 0 and (size-1) will not be claimed by the maze. Columns 0 and (size-1) will not be claimed by the maze.

Each thread should start in a different corner of the array. They will need to start in location (1,1), (1, (size-2)), ((size-2), 1), and ((size-2), (size-2)). The serial program should show only one thread (rank 0).

If an even-sized array is used, then the last row and column won't be claimed. The algorithm grows in steps of two, which means that it cannot claim a single row or column. Use odd-numbered array sizes to fill the entire maze.

It will be unlikely that all four processors will claim equal amounts of the space, unless a very large array is used. It is normal for threads 2 and 3 to claim little of the space. To test that the system is working you can add a sleep after the critical section which will allow each thread an opportunity to execute. Remember to **remove the sleep from your code** before you submit it for grading.

When adding the OpenMP directives it is important that each thread runs a separate copy of the search algorithm. The same function which performs the search should be executed by each thread. There should not be a different function for each thread. Only the parameters passed to the function should change.

The parallel version of the program should always run four threads.

Only one thread should be able to test its neighbours or claim space at a time or a race condition will occur.

When the maze is complete, print out the number of times each thread claimed a neighbour. See the A3 overview lecture notes for some sample output.

Command Line Arguments
The program should accept `-n` to indicate the size of the array. For example:

```
maze -n 21
```

would create an array that is 21 by 21 elements. If no `-n` argument is used then make the array 11 by 11.

The `-s` argument is used to seed the random number generator for determining the order in which the neighbours are visited. If no `-s` argument is used, seed the random number generator with 0.

Documentation
Include a readme.txt contains your name and student number. If anything does not work then you should mention it in the file.

Coding Practices
Write your assignment in C using the OpenMP library.

Write the code using standard stylistic practices. Use functions, reasonable variable names, and consistent indentation. If the code is difficult for the TA to understand then you will lose marks.

As usual, keep backups of your work using source control software.

Submission and Evaluation

Submit the assignment using Moodle. Submit only the source code and the makefile. Bundle the code in a zip file.

The assignments will be marked on the `linux.socs.uoguelph.ca` server. If you develop your code on a different platform then it is a good idea to put the include files in an #ifdef for that system so they will still compile on the server. Test your program on the SoCS Linux server before you submit it.

The TA will unpack your code and type "`make`". It must produce tow executable:

- `maze`, which implements the serial maze generation.
- `mazep`, which implements the parallelized maze generation.

If the makefile is missing, the make command does not work, or the program does not execute then you will lose a substantial number of marks.

It is always a good idea to unpack and test the file you are submitting to be sure that what you submit actually compiles.