Christian Loera

CSC 462 – C++ Multithreading

Maze Writeup

For my algorithm, I decided to use a modified version of depth-first search spread across 6 different threads. I realized that the best way to tackle the mazes was to essentially divide and conquer, spreading the work amongst all threads. So instead of searching from only one side of the maze, my approach was to search both ends. Since the start and end of the maze are predetermined, I start my maze searching at the entrance and exit of the maze and move towards the opposite ends of the maze until I hit a location that has been visited by another thread. When my searching threads have found a collision they each signal their own path tracing threads to recover the path back to their starting points. That would leave me with two lists which would then be combined by my main thread to create the full path solution. This strategy would have been extremely slow if it were not for depth-first search.

When I first started this project, I began with a breadth-first search algorithm, but quickly realized that it would be extremely difficult to get the speed that I needed even if it was multithreaded. That's when I opted to use the depth-first search since it searches deep before searching wide. My depth-first search is slightly different than common implementations in that it traverses through the maze until it hits a junction or a dead-end, at which point it will push it onto a list. This prevents multiple node allocations on the list, which can be costly. The current position being processed must have at least one possible direction to move towards or else it will be popped from the list and the previous junction will then be processed. If a position has been marked by some other thread then the last marked position will be stored, the path tracing thread will be notified, and the function will immediately exit.

At the very beginning of my program I allocate my three objects on the heap in main(), DFSTop, DFSBottom, and StudentSolveBuilder. Each of these classes inherit from the provided MTMazeStudentSolver class so they require a pointer to Maze to be passed in. The DFSTop and DFSBottom objects are used to explore the maze using depth-first search and find half of the solution path. The StudentSolveBuilder object, studentSolver, is used combine both halves of the solution into the final solution that would be stored in *pSolutionStudent, which is done when studentSolver calls the Solve() function. Within Solve() two lists of type Direction are allocated on the heap and are then passed as parameters, each to different threads. These threads are also allocated on the heap and call DFSTop and DFSBottom's partialSolve() functions. Each of these threads also take in a reference of the DFSTop and DFSBottom objects that were created in main() and passed into StudentSolveBuilder as arguments during construction. Thread t1 calls DFSTop's partialSolve and takes topPath list as a reference, while Thread t2 calls DFSBottom's partialSolve and takes bottomPath as a reference argument. Both threads are then joined to ensure that both lists have been filled with Directions and can be combined.

Inside DFSTop and DFSBottom's partialSolve() functions, they each create 2 threads allocated on the heap and call their own implementations of DFS() and TracePath() functions. DFS() is where the mazes are processed with a depth-first search algorithm. In DFSTop, the DFS() function traverses the maze from top to bottom until it finds a position visited by DFSBottom. DFSBottom's DFS() function mirrors DFSTop's DFS() function in that it starts at the end of the maze and traverses upwards until it visits a position that has already been visited by DFSTop. TracePath() is where Directions are pushed onto a list based on the processed path in DFS(). The list used in TracePath() is the same list that was passed as reference into t1 and t2 threads in StudentSolveBuilder. While the threads that called DFS() and TracePath() functions are running partialSolve() joins the threads, waiting for them to finish running. In total there are 2 threads in DFSTop's partialSolve(), 2 threads DFSBottom's partialSolve(), and 2 threads in StudentSolveBuilder's Solve() function.

With all these threads, you would think there would be lots of communication between threads, but in fact there is very little. In MTMazeStudentSolver I have 2 atomic bools used for signaling TracePath() functions when they can begin finding the solution path. The atomic bool isDFSTopDone is used to signal DFSTop::TracePath() that the DFSTop::DFS() function has found a cell in the maze marked as visited by DFSBottom::DFS(). The signal is done simply by setting isDFSTopDone to true atomically by using store(). The atomic bool isDFSBotDone is used the exact same way, but for DFSBottom. So when DFSBottom::DFS() finds a cell marked by DFSTop it will set isDFSBotDone to true, letting DFSBottom::TracePath() know to start finding the solution path. The TracePath() functions wait for their signals by checking if their respective atomic bools are set to true within a while loop. So essentially it uses a spin lock to wait for a signal. I chose to use this method instead something else like a conditional variable because I ran into a lot of deadlocking problems, which I could not resolve. I suspect this had to do with the architecture of my system and would have required me to completely rethink my approach.

In my application, I decided to replace the maze's ints with atomic ints, so that I could modify the maze data without having to lock it and take hits to my performance. The atomics made it easier for me to freely modify the data, allowing me to use the entire int of each cell to mark as visited or mark the direction in which I came from. This was important to my application because I can pretty much recover the searched path or check for a collision using atomic operations. The MazeMarker enum class also made it easier for me know what bits were being operated and helped with readability.

One of the hardest parts of this assignment was figuring out how I wanted to set up the architecture of this application. I knew that if I set up my system in such a way that every thread did something then I could have gotten the most out of my performance. My implementation was very dependent on the status of other threads, which caused some threads to wait unnecessarily. This waiting seemed to be caused by the fact that I essentially created two single threaded programs that ran in parallel, even though I used six threads in my application. This might have been because I joined all my threads and performed operations that depended on the joins to finish. What made this even more difficult was that I had to implement and call the Solve() function in some way, which somewhat limited me in the ways I could have designed my application. Architecting a multithreaded application was probably the hardest lesson that I've learned since it effected what multithreading tools I could have used and the overall performance of my system.