Indian Institute of Technology, Delhi

Parallel and Distributed Programming (COL 380)

Prof. Rijurekha Sen

Assignment 3

Mani Sarthak (2021CS10095)
Harshit Gupta (2021CS10552)
Rishabh Verma (2021CS10581)

May 8, 2024

Contents

§1 Parallel Maze Generation

Despite the code's generic nature, let's focus on generating a maze of fixed size 64 * 64 using 4 processors as an example. We begin with a 32 * 32 matrix, where each element represents a node in a graph. Connectivity is established such that only neighboring nodes in the matrix are connected in the graph, with random weights assigned to the edges. Next, we generate the Minimum Spanning Tree (MST) of the graph using both Breadth-First Search (BFS) and Kruskal's algorithm.
The resulting MST possesses two key properties:

- The randomness of edge weights yields a random MST.
- The MST comprises only edges adjacent to the matrix elements.

Converting the MST into a perfect maze leverages the MST concept effectively. Firstly, it facilitates the creation of a random maze. Secondly, as an MST forms a tree, it ensures a single path between any two nodes, thereby creating a perfect maze. To translate the MST into a maze, we assign a value of 1 to nodes connected by an edge and 0 otherwise. This process reduces the matrix size to 63*63, which we then expand by adding a column and a row to attain the desired 64 * 64 size.

The sequential maze generation process transitions seamlessly into parallelization. We exploit the independence of 4 processors to generate 4 MSTs, each corresponding to a processor. By halving the dimensions of MSTs requested from each processor and mapping the nodes accordingly, we ensure consistency in the larger dimension. Connecting any three edges between the unique MSTs ensures overall graph connectivity. Subsequently, generating the larger MST and applying the maze generation scheme yields a random perfect maze.

§2 Solving maze parallely

Just like in the above approach of building a generator in solving the maze. In simple terms, we have a 64x64 matrix which basically contain 4 MSTs divided in 4 grid blocks of 32x32. So in our approach simply first created a serial code which takes a maze as parameter as well as start and end node (here we are talking general start and end nodes rather than top-right and bottom-left).
So we created a serial code for finding a path using **dfs** as well as **dijkstra** algorithm. after that we allocated a master as well as slaves processes broadly 0 is the master process and 1,2,3 are the slaves processes. so a master process first take a input and broadcast it to the slaves as well. after that we simply apply algorithms to the the portioning grids to find the path after that we club the results and we get the final paths.

§3 Description of algorithms

§3.1 Breadth-First Search (BFS)

In the `bfs.cpp` file, we implemented Prim's algorithm to construct the Minimum Spanning Tree (MST) of the graph. The edges belonging to the MST are utilized to generate the maze following the previously described scheme.

The `createMST_BFS` function takes the graph's adjacency matrix as input and employs Prim's algorithm to generate the MST in the form of an adjacency list. Additionally, the `getMST_BFS` function abstracts the process of obtaining an adjacency list representation of the MST.

§3.2 Kruskal's Algorithm

In the `kruskal.cpp` file, Kruskal's algorithm is applied to determine the MST of the graph, which is subsequently utilized in the `mazegenerator.cpp` to create the maze.

We utilized the Disjoint Set Union data structure to compute the MST. The `kruskal_MST` function computes the MST of a graph in adjacency list format based on its weighted edges. Additionally, the `getMST_Kruskal` function abstracts the process of obtaining the MST of a weighted graph in the form of an adjacency list using Kruskal's algorithm.

§3.3 Depth-First Search (DFS)

In the depth-first search (DFS) segment of the solver code, our approach in the serial implementation involved maintaining a stack. We initially added the starting node to the stack and commenced a while loop.

In each iteration, we popped the top element from the stack. We then checked if it had been traversed before, which was represented by a matrix of size 64x64 where all cells were initialized to 0 except those along the traversed path. If we encountered a cell again in the loop, indicating backtracking, we removed it from the path and continued.

If the node was not previously visited, we added it to the path, marked it as visited, pushed it back onto the stack, and proceeded to check its neighboring nodes. Valid neighbors that were unvisited and unblocked were added to the stack.

In the parallel version, we followed a similar process but for smaller grids, and subsequently combined the paths from different processes.

§3.4 Dijkstra's Algorithm

In the context of solving a random perfect maze, where edges have a uniform weight of 1, Dijkstra's algorithm simplifies to a **Breadth-First Traversal**. Instead of a priority queue, a simple queue is employed to manage the exploration frontier.

Key Points:

- **Simplified Approach**: With uniform edge weights, Dijkstra's algorithm essentially reduces to Breadth-First Traversal.
- **Queue Utilization**: A basic queue suffices for handling the frontier, eliminating the need for a priority queue.
- **No Visited Cell Storage**: Due to tracking distances from the source and the single-path nature of the maze, storing visited cells is unnecessary.

In the parallel implementation, two sets—frontier and next frontier—are maintained. The next frontier comprises nodes reachable from the current frontier, with distances updated iteratively. `MPI_Reduce` is used to compute minimum distances, and local next frontiers are merged after each iteration for synchronization.

Key Points:

- **Frontier Management**: Utilizing frontier and next frontier sets optimizes exploration and expansion.
- **MPI_Reduce Usage**: `MPI_Reduce` facilitates minimum distance computation, with requests sent for distances across process partitions.
- **Local Frontier Merge**: Merging of local next frontiers post-iteration ensures synchronization and progress.

§4 Efficiency Analysis

§4.1 Generator

The main components include:

1. Initializing the graph for MST ($n \times n$ dimensions) – $O(n^2)$.
2. Processing the graph to prepare it for algorithm input – $O(n^4)$ for BFS adjacency matrix, $O(n^2)$ for Kruskal.
3. Executing the algorithm – $O(V + E)$ or $O(n^2)$ for BFS, $O(E \log E)$ or $O(n^2 \log n)$ for Kruskal.
4. Mapping vertices as if they were on the expanded plane – $O(n^2)$.

These processes run independently on processors. Now, transitioning to the sequential code:

1. Sending data to Processor 0 – Note: Only edges are sent, as transmitting the adjacency matrix would severely impact performance. Moreover, sending in the adjacency list representation would require the use of derived datatypes and introduce complexity. Hence, edge transmission is considered optimal, requiring $O(t_w \times n^2 \times p)$ time.
2. Merging the MSTs – $O(n^2)$ time.
3. Finally, creating the grid – $O(n^2)$ time.

§4.2 Solver

Analysis of Algorithms:

1. We consider an $n \times n$ grid.
2. The serial depth-first search (DFS) takes $O(n^2)$ time to traverse the entire grid.
3. For the parallel implementation, we divide the grid into 4 parts and run DFS concurrently on each partition. Thus, each grid takes $O(n^2/p)$ time, where p is the number of processors.
4. Therefore, the time complexity for parallel DFS is $O(n^2/p) + O(w)$, where $O(w)$ represents the overhead for parallel communication.
5. Similar analysis applies to Dijkstra's algorithm. Here, the time complexity is $O(n^2/p)+O(w)$ due to the overhead of parallel communication. The $O(n^2/p)$ component arises because we are dealing with an unweighted graph, which reduces the problem to a breadth-first search (BFS), thus resulting in this time complexity.

§4.3 Notes on measurement

To calculate the overall speedup, we need the execution times of the serial and parallel implementations of both the generating and solving algorithms. Let's denote these as follows:

For generating algorithms: - $T_{\text{serial\_generate}} = T_{\text{serial\_maze\_generation}}$ - $T_{\text{parallel\_generate}} = T_{\text{parallel\_maze\_generation}}$

For solving algorithms: - $T_{\text{serial\_solve}} = T_{\text{serial\_maze\_solving}}$ - $T_{\text{parallel\_solve}} = T_{\text{parallel\_maze\_solving}}$

Substituting the given values into the speedup formulae, we have:

$$S_{\text{generate}} = \frac{T_{\text{serial\_generate}}}{T_{\text{parallel\_generate}}} = \frac{T_{\text{serial\_maze\_generation}}}{T_{\text{parallel\_maze\_generation}}}$$

$$S_{\text{solve}} = \frac{T_{\text{serial\_solve}}}{T_{\text{parallel\_solve}}} = \frac{T_{\text{serial\_maze\_solving}}}{T_{\text{parallel\_maze\_solving}}}$$

Finally, the overall speedup is given by the product of these two speedups:

$$S_{\text{overall}} = S_{\text{generate}} \times S_{\text{solve}} = \frac{T_{\text{serial\_maze\_generation}}}{T_{\text{parallel\_maze\_generation}}} \times \frac{T_{\text{serial\_maze\_solving}}}{T_{\text{parallel\_maze\_solving}}}$$

We can plug in the actual values of the execution times to calculate the overall speedup.

Efficiency (E) can be calculated as the ratio of the speedup (S) to the number of processors (p). Here's the formula:

$$E = \frac{S}{p}$$

Substituting the expression for speedup into this formula, we get:

$$E = \frac{S_{\text{generate}} \times S_{\text{solve}}}{p}$$

$$E = \frac{T_{\text{serial\_maze\_generation}}}{T_{\text{parallel\_maze\_generation}}} \times \frac{T_{\text{serial\_maze\_solving}}}{T_{\text{parallel\_maze\_solving}}} \times \frac{1}{p}$$

This formula gives us the efficiency of the parallel implementation, which indicates how effectively the parallel resources are utilized compared to the ideal case.

§5 MPI Synchronization

- In `maze.cpp`, the maze-solving process utilizes functions from `mazesolver.cpp` immediately after generating a random perfect maze from `mazegenerator.cpp`.
- All of these tasks are encapsulated within a single framework of **MPI_Init** and **MPI_Finalize**. Given that each processor in our implementation generates a subgrid of size 32x32, it's possible for one processor to finish its maze generation and start solving the maze while others are still generating their subgrids.
- To ensure proper synchronization and prevent premature maze solving, we employ **MPI_Barrier** after maze generation. This ensures that all processors complete maze generation before any process attempts to solve the maze.

§6 MPI Reduction

- **MPI_Reduce** is utilized to merge the distances of nodes from the source.
- After each iteration of Dijkstra's algorithm, each process possesses partial distance information. **MPI_Reduce** aggregates these partial distances across all processes.
- Following the reduction operation, the root process consolidates the final shortest path distances for all vertices. These distances represent the shortest paths from the source vertex to all other vertices in the graph.

§7 Optimizations

Given that Minimum Spanning Trees (MSTs) are utilized, the graphs involved tend to be very sparse, with the number of edges ($|E|$) being equal to the number of vertices minus one ($|V| - 1$). Consequently, when communicating the MSTs, only the number of edges needs to be sent, significantly reducing overhead.

Additionally, as we are dealing with smaller data sets for message passing, we opt for simple communication methods such as `MPI_Send` and `MPI_Recv`. Notably, `MPI_Send` operates as a non-blocking method, offering optimization opportunities. Moreover, minimizing the amount of data sent remains the most straightforward approach to reducing communication costs.

Throughout the process, communication occurs at the following key points:

1. Gathering individual MSTs to generate the grid.
2. Partitioning the grid to independently solve the maze on each processor.
3. Finally, reducing the individual paths generated by processors and printing the output.

These optimizations contribute to enhancing the efficiency and performance of the maze-solving process.