Creating and solving random maze with graph algorithms on distributed memory

Nupur Unavekar

Files Submitted

```
kerbno1_kerbno2_kerbno3
├── maze.cpp
├── generator
│   ├── mazegenerator.hpp
│   ├── bfs.hpp
│   ├── kruskal.hpp
│   ├── mazegenerator.cpp
│   ├── bfs.cpp
│   └── kruskal.cpp
└── solver
    ├── mazesolver.hpp
    ├── dfs.hpp
    ├── dijkstra.hpp
    ├── mazesolver.cpp
    ├── dfs.cpp
    └── dijkstra.cpp
```

Commands to Run:

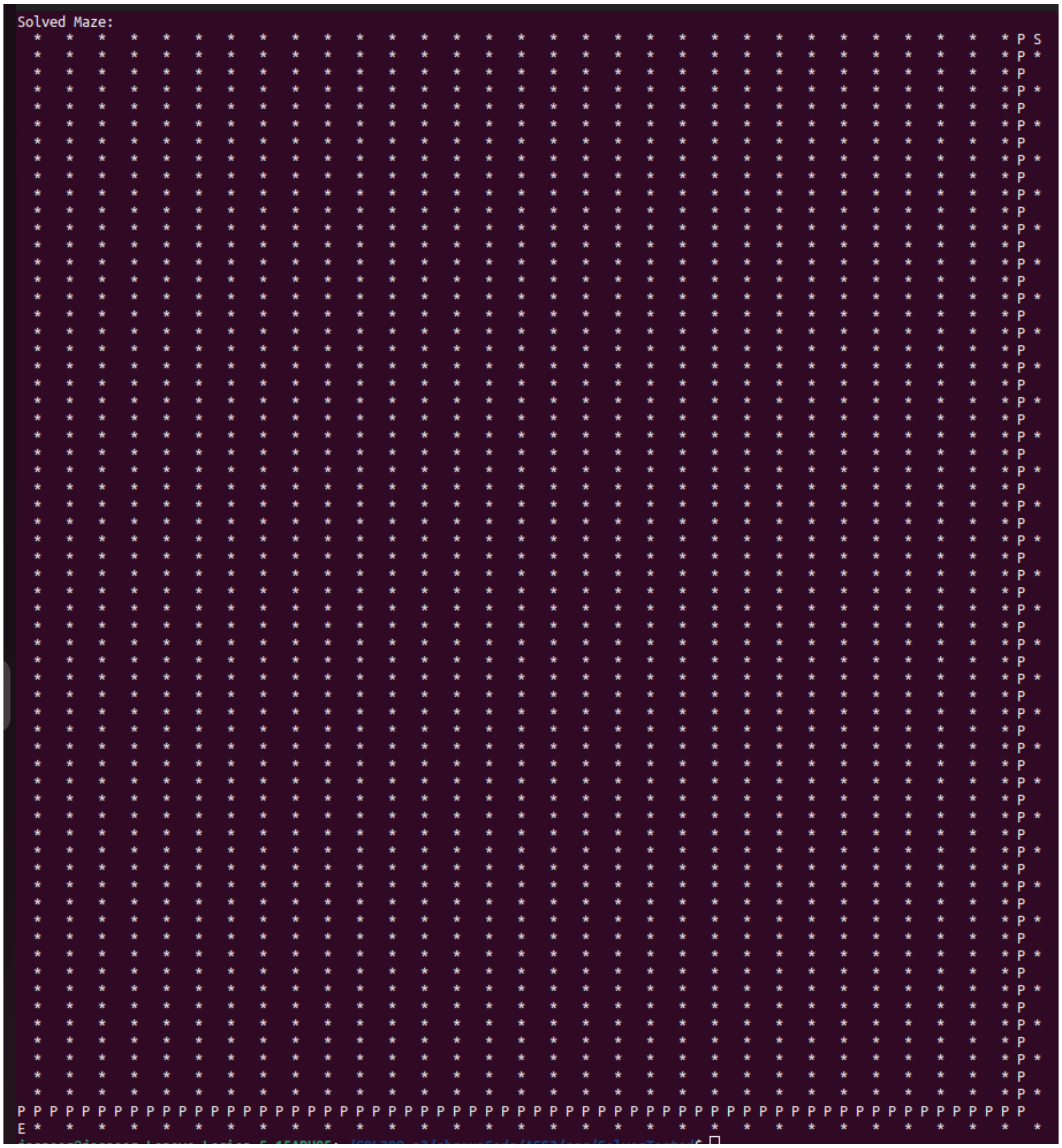
- make: Compile all the files required
- Enter the command to run the algorithms, example: mpirun -np 4 ./maze.out -g [bfs/kruskal] -s [dfs/dijkstra]
- 'make clean' to clear the build

How did we generate perfect maze using graph algorithms?

In this report, we present a novel approach to generating perfect mazes using parallel Breadth-First Search (BFS) and Kruskal's algorithm. By employing parallelism, we efficiently construct tree structures within the graph, which serve as the foundation for maze generation. Utilizing BFS, we ensure orderly exploration of the graph, while Kruskal's algorithm facilitates the creation of a minimum spanning tree based on edge weights. These trees are then transformed into mazes, leveraging their unique path properties to guarantee a maze layout with distinct paths between any two vertices. This approach offers a robust solution for generating intricate mazes with parallel computing techniques, ideal for various applications requiring maze generation with unique path configurations. We will learn more about actuall parallel implementation of these algorithms i n the comming sections

How did we solve the random maze using graph algorithms?

According to the problem statement, we need to trace and mark the path from source 'S' (top-right cell) to end 'E' (bottom-left cell). The wall cells in the path have '*', the cells along the solution path have 'P' and 'S' for start cell and 'E' for end cell. We use 2 graph algorithms DFS and Dijkstra algorithm, to solve the maze and mark the path for output. The solver folder contains the implementation of these algorithms. With dfs algorithm we start with marking the start cell as S. Next we perform the dfs along any path in it's neighbour which is not a wall. If at last we find the destination cell then return true otherwise perform dfs in any other neighbour direction. As we move along the path we keep marking 'P' (erase if not in the solution path). Using Dijkstra, we follow the algorithm taught in class, single source shortest path using priority queues



```
Solved Maze:
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P S
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P
  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   * P *
P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P P
E *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *
```

Figure 1: 64*64 maze solution path

MPI Implementation Details

We begin by establishing a Fully Connected Tree (FCT) comprising n nodes, with the input terminal serving as the root node. This FCT serves as the foundation for our subsequent operations.

The next step involves utilizing the Breadth-First Search (BFS) algorithm to create a tree structure through Message Passing Interface (MPI) protocols. This process entails distributing the nodes of the FCT among four processors for parallel execution.

The parallelization methodology hinges on concurrently constructing the BFS tree across multiple processors. This tree serves as the framework for generating a maze that adheres to the criteria of a perfect maze, wherein every pair of points possesses a unique path between them.

Elaboration:

- **Fully Connected Tree (FCT):** This is a tree data structure where every node is connected to every other node in the tree. In the context of the described process, the FCT serves as the initial structure from which the maze will be generated.
- **Breadth-First Search (BFS):** BFS is an algorithm used for traversing or searching tree or graph data structures. In this context, BFS is applied to create a tree structure from the Fully Connected Tree.
- **Message Passing Interface (MPI):** MPI is a standardized and portable message-passing system designed to facilitate communication between multiple processors or computing nodes. It is commonly used in parallel computing environments.
- **Parallelization:** The process of parallelization involves dividing a task into smaller parts that can be executed simultaneously across multiple processors or computing nodes. In this case, the BFS algorithm is parallelized by distributing the nodes of the Fully Connected Tree among four processors, allowing for faster execution and scalability.
- **Perfect Maze:** A perfect maze is a maze in which there is exactly one path between every pair of points. By utilizing the BFS tree generated through parallel processing, the resulting maze will adhere to this criterion, ensuring that every two points in the maze are connected by a unique path.

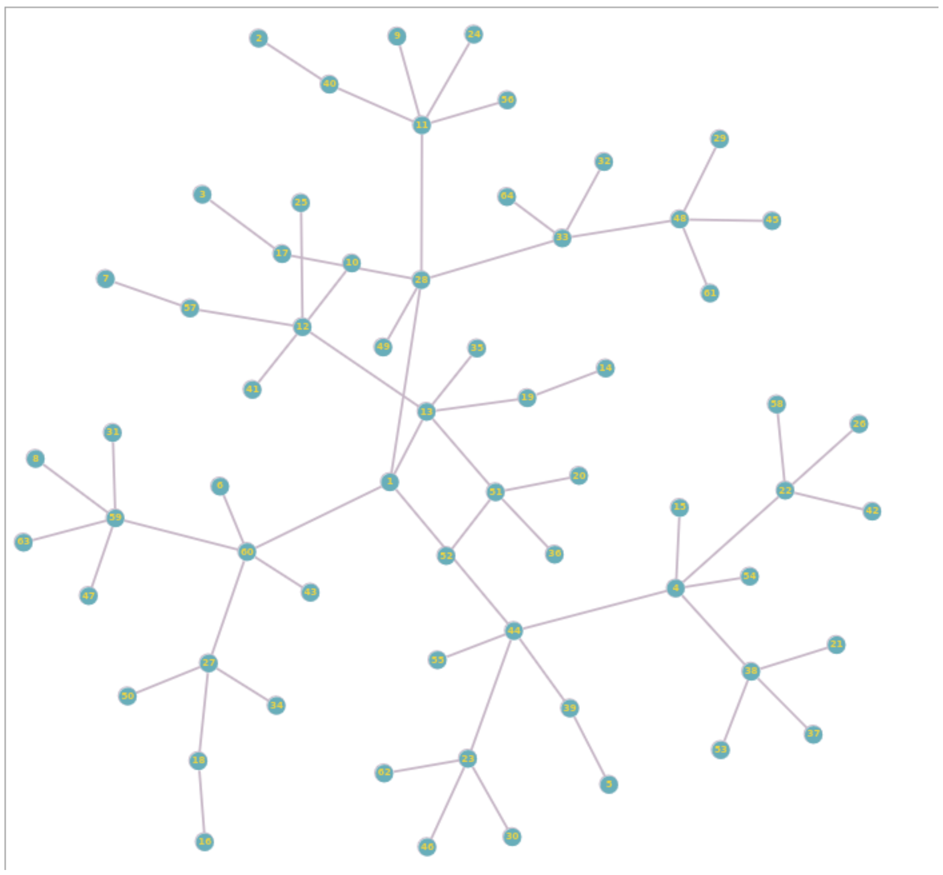


Figure 2: 64*64 maze solution path

MPI Collective calls used

- **MPI_Scatterv**: This call is being used to distribute the adjacency matrix of a graph to the processors. Each processor will work only on the respective local matrix.

```
void distribute_graph(int* adj_mat, int n, int rank, int size, int*
    local_adj_mat){
    int* sendbuf =  (int*)malloc(n*n*sizeof(int));
    int* sendcounts = (int*)malloc(size*sizeof(int));
    int* displs = (int*)malloc(size*sizeof(int));

    // distribute graphs based on vertices

    int num_vertex_per_process = n/size;

    for(int i=0; i<size; i++){
        sendcounts[i] = num_vertex_per_process*n;
        displs[i] = i*num_vertex_per_process*n;
    }

    MPI_Scatterv(adj_mat, sendcounts, displs, MPI_INT, sendbuf,
        num_vertex_per_process*n, MPI_INT, 0, MPI_COMM_WORLD);

    for(int i=0; i<num_vertex_per_process; i++){
        for(int j=0; j<n; j++){
            local_adj_mat[i*n+j+displs[rank]] = sendbuf[i*n+j];
        }
    }

}
```

Listing 1: MPI_Scatterv

- **MPI_Alltoallv**: This call is being used as: Alltoallv(Buffer, RecvBuff). Each process is sending some data corresponding to vertices to their respective owner, as well as receiving from other processors as well.

```
int* recv_buff = (int*)malloc(size*n*n*sizeof(int));
        MPI_Alltoallv(to_send, sendcounts, displs, MPI_INT, recv_buff,
            sendcounts, displs, MPI_INT, MPI_COMM_WORLD);
        // process the received data into next_frontier removing duplicates
        for(int i=0; i<size; i++){
            for(int j=0; j<n*n; j+=2){
                if(recv_buff[i*n*n + j] == -1){
                    break;
                }
                next_frontier.push_back(make_pair(recv_buff[i*n*n + j],
                    recv_buff[i*n*n + j + 1]));
                // printf("Rank %d: %d %d\n", rank, recv_buff[i*n*n + j],
                    recv_buff[i*n*n + j + 1]);
            }
        }

        // remove entries with same i and randomply keep one
        random_shuffle(next_frontier.begin(), next_frontier.end());
        // keep the first occurence of each parent neighbour pair

        MPI_Barrier(MPI_COMM_WORLD);
```

Listing 2: MPI_Alltoallv and MPI_Barrier

- **MPI_Allreduce**: Used for checking the termination condition. It reduces the sizes of all frontiers and distributes the value of size of global frontier to all the processors. If the size is 0, this means that the computation is complete and all processors exit the while loop.

```
        int size_frontier = current_frontier.size();
        int haha = size_frontier;
        MPI_Allreduce(&size_frontier, &haha, 1, MPI_INT, MPI_SUM,
           MPI_COMM_WORLD);
```

Listing 3: MPI_AllReduce

MPI reductions

- **MPI_Bcast**: This broadcasts the value of size of matrix to all the processors initially.

```
    if(rank == 0){
        n = int(atoi(argv[1]));
        adj_mat = create_adj_mat(n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Listing 4: MPI_Bcast

Use of Synchronization Primitives for Correctness

In the provided code, `MPI_Barrier(MPI_COMM_WORLD)` is used to ensure synchronization among processes. This barrier ensures that no process proceeds until all processes have reached the same point in the code. It's often used to ensure that all processes have completed their computation up to a certain point before proceeding further. In this case, it is used after processing each level of the BFS traversal.

MPI Blocking vs. Non-blocking Calls

The code primarily uses blocking MPI calls, such as `MPI_Alltoallv` and `MPI_Reduce`. Blocking calls suspend the execution of the calling process until the communication operation completes. This ensures that data is correctly transmitted before the process proceeds. Non-blocking calls, like `MPI_Isend` and `MPI_Irecv`, are not used in the provided code. These calls allow a process to initiate communication and continue its execution without waiting for the communication to complete. However, they require careful management of buffers and communication statuses to ensure correctness.

MPI Reductions Used

The code utilizes an MPI reduction operation with `MPI_Reduce` to merge the BFS tree adjacency matrices from all processes into a single final matrix. The reduction operation sums the corresponding elements of the matrices, effectively accumulating the results from all processes. This is essential for constructing the complete BFS tree.

Optimizations for Handling Sparsity of the Maze Graph

In a maze graph, each node typically has a constant number of neighbors, resulting in a sparse adjacency matrix. To optimize memory usage and computation time, the code only stores and processes the non-zero entries of the adjacency matrix. Additionally, the code avoids duplicate entries in the BFS traversal frontier by keeping track of visited nodes and discarding redundant edges. This optimization reduces unnecessary computation and communication overhead.

1 Kruskal Implementation

Similar to BFS we are working with Kruskal first my assigning random weights to fully connected graph and then using that graph to find the MST. Vsrious Functions involved are -

Scatter Edge List (scatterEdgeList)

```c
void scatterEdgeList(int* edgeList, int* edgeListPart, const int elements, int*
    elementsPart) {
    int rank;
    int size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    MPI_Scatter(edgeList, *elementsPart * 3, MPI_INT, edgeListPart, *elementsPart
        * 3, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank == size - 1 && elements % *elementsPart != 0) {
        *elementsPart = elements % *elementsPart;
    }

    if (elements / 2 + 1 < size && elements != size) {
        if (rank == 0) {
            fprintf(stderr, "Unsupported size/process combination, exiting!\n");
        }
        MPI_Finalize();
        exit(EXIT_FAILURE);
    }
}
```

Listing 5: Scatter Edge List function

This function scatters the edge list among MPI processes. It uses MPI_Scatter to distribute the edge list to each process. It adjusts the number of elements each process receives based on the total number of elements and the number of processes. It ensures that the last process receives the remaining elements if the total number of elements is not evenly divisible by the number of processes.

Merge Sort (mergeSort)

This function implements the merge sort algorithm to sort the edge list. It recursively divides the edge list into smaller parts until each part contains only one element. It then merges adjacent parts in sorted order. It uses the merge function to merge two sorted parts of the edge list.

Merge (merge)

This function merges two sorted parts of the edge list into a single sorted part. It creates a working array to store the merged result. It compares elements from the two parts and copies them to the working array in sorted order. Finally, it copies the merged result back to the original edge list.

```c
void join(int* edgeList, const int start, const int end, const int pivot) {
    int length = end - start + 1;
    int* working = (int*) malloc(length * 3 * sizeof(int));

    memcpy(working, &edgeList[start * 3], (pivot - start + 1) * 3 * sizeof(int));

    int workingEnd = end + pivot - start + 1;
    for (int i = pivot + 1; i <= end; i++) {
        copyEdge(&working[(workingEnd - i) * 3], &edgeList[i * 3]);
    }

    int left = 0;
    int right = end - start;
    for (int k = start; k <= end; k++) {
        if (working[right * 3 + 2] < working[left * 3 + 2]) {
            copyEdge(&edgeList[k * 3], &working[right * 3]);
            right--;
        } else {
            copyEdge(&edgeList[k * 3], &working[left * 3]);
            left++;
        }
    }

    free(working);
}
```

Listing 6: Join Function

Sort (sort)

```
void Msortt(int* edgeList, const int start, const int end) {
    if (start != end) {
        int pivot = (start + end) / 2;
        Msortt(edgeList, start, pivot);
        Msortt(edgeList, pivot + 1, end);

        join(edgeList, start, end, pivot);
    }
}
```

Listing 7: Msortt Function

This function orchestrates the sorting process for the entire edge list. It broadcasts the total number of elements to all processes. It determines the number of elements each process will receive and scatters the edge list accordingly. It performs a parallel merge sort by recursively merging sorted parts received from other processes. It handles communication between processes using MPI_Send and MPI_Recv.

MST Kruskal Algorithm (mstKruskal)

```
void mstKruskal(WeightedGraph* graph, WeightedGraph* mst) {
    Set tempSet = { .elements = 0, .canonicalElements = NULL, .rank = NULL };
    Set* set = &tempSet;
    newSet(set, graph->vertices);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    sort(graph);

    if (rank == 0) {
        int currentEdge = 0;
        for (int edgesMST = 0; edgesMST < graph->vertices - 1 || currentEdge <
            graph->edges;) {
            int canonicalElementFrom = findSet(set, graph->edgeList[currentEdge *
                3]);
            int canonicalElementTo = findSet(set, graph->edgeList[currentEdge * 3
                + 1]);
            if (canonicalElementFrom != canonicalElementTo) {
                copyEdge(&mst->edgeList[edgesMST * 3],
                    &graph->edgeList[currentEdge * 3]);
                unionSet(set, canonicalElementFrom, canonicalElementTo);
                edgesMST++;
            }
            currentEdge++;
        }
    }

    dltSt(set);
}
```

Listing 8: mstKruskal Function

This function implements the Kruskal algorithm to find the Minimum Spanning Tree (MST) of a weighted graph. It first sorts the edge list using the sort function. It then iterates through the sorted edge list, adding edges to the MST if they do not create a cycle. It uses a disjoint-set data structure to efficiently check for cycles and merge disjoint sets.

Use of Synchronization Primitives for Correctness:

In the provided implementation, synchronization primitives are not explicitly used. However, correct synchronization is implicitly achieved through MPI communication calls. For example, `MPI_Comm_rank` and `MPI_Comm_size` are used to determine the rank and size of each process, ensuring correct communication and synchronization among processes.

MPI Blocking vs. Non-blocking Calls Used:

The implementation primarily uses blocking MPI calls. For instance, `MPI_Scatter`, `MPI_Recv`, and `MPI_Send` are blocking communication calls that ensure processes wait until the communication operation completes before proceeding. Blocking calls are suitable here as they provide a simple and straightforward way to manage communication between processes and ensure data integrity.

MPI Reductions Used:

No MPI reduction operations are used in the provided Kruskal's algorithm implementation. Reduction operations are typically used to perform a collective operation where data from all processes are combined using an operation like sum, max, min, etc. While Kruskal's algorithm itself doesn't inherently require reduction operations, they could potentially be used for certain optimizations or analysis depending on the specific requirements.

Optimizations for Handling Sparsity of the Graph:

The implementation does not explicitly optimize for the sparsity of the graph. However, Kruskal's algorithm naturally handles sparse graphs efficiently. Since Kruskal's algorithm processes edges in non-decreasing order of weight and checks for cycles, it inherently skips over edges that would create cycles in the MST. This characteristic makes it well-suited for sparse graphs, where the majority of edge weights are likely to be higher, resulting in fewer edges being included in the MST.

Additionally, the implementation sorts the edge list using merge sort before processing, which can be efficient for sparse graphs as it has a time complexity of $O(E \log E)$, where E is the number of edges. In sparse graphs, E is typically much smaller than the total number of possible edges, leading to efficient sorting.

Tree to Maze Conversion Function

2 Speedup and efficiency

- **BFS**: The sequential code had a time complexity of O(V+E) where V is the number of vertices and E is the number of edges hence a time complexity of O(N^2) (N is the number of vertices). With MPI parallelization, the time complexity reduced down to O($\frac{E}{4}$).