Distributed Systems HW-3 (MPI)

Q1

Main Idea :-

As we need to take input from one spawned process, the root process (rank 0) reads the number of points (`N`), number of queries (`M`), and the number of nearest neighbours (`K`) and broadcasts these to all other processes. The root process also reads the coordinates of all points and queries and The points and queries are then broadcast to all processes using `MPI_Bcast` to ensure consistency across processes.

Each process is assigned a subset of the queries to handle. The queries are distributed such that each process handles approximately ⌈M/p⌉ queries. This distribution ensures that the workload is balanced among the processes.

For each query assigned to a process, the process computes the Euclidean distance from the query point to all points in the set. The distances are partially sorted, and the K nearest points are identified. This step is repeated for all queries handled by the process.

Each process sends its results (K nearest neighbors for each query it handles) to the root process using `MPI_Gather`. The root process collects all the results and organizes them according to the order of the queries . And finally, the root process prints the K nearest neighbours for each query in the order of the queries.

Q2

Main Idea :-

As we need to take input from one spawned process(i.e here process with rank 0), the root process (rank 0) reads the grid dimensions (`N`, `M`), iteration count (`K`), and the constant complex number `c` from the input. The grid dimensions and constant `c` are broadcast to all processes using `MPI_Bcast` to ensure that all processes have the same information.

The total number of grid points is `N * M`. The array of grid points is distributed among the processes.

1. Since `p` (number of processes) might not perfectly divide `N * M`, the extra points (`p % (N * M)`) are distributed among the first `p % (N * M)` processes to balance the load.
2. Each process handles either the floor or ceiling of `N * M / p` elements, ensuring a balanced workload.

Each process computes whether its assigned grid points belong to the Julia Set.For each grid point, the function zn+1=zn2+c is iterated up to `K` times or until the magnitude of `z` exceeds the threshold `T = 2`. If the magnitude does not exceed `T` within `K` iterations, the point is considered to be part of the Julia Set.

After computing the Julia Set membership for its assigned points, each process sends its results to the root process using `MPI_Gatherv`. The root process collects all the results and constructs the final output grid.

Q3

Main Idea :-

As we need to take input from one spawned process (i.e here process with rank 0) , we broadcasted the size of the array `n` to all processes using `MPI_Bcast` to ensure all processes are aware of the total number of elements. The array is distributed equally among the processes using `MPI_Scatterv`. Each process receives a portion of the array, which is computed based on the number of elements each process is responsible for (`send_counts`) and their respective starting positions (`displs`).

Each process computes the prefix sum locally on its portion of the array. This is done by iterating through the local segment and adding each element to the sum of all preceding elements in the local array. After computing the local prefix sums, the results are gathered back to the root process using `MPI_Gatherv`. Each process sends its computed prefix sum data to the root process.

The root process adjusts the prefix sums gathered from all processes to ensure they represent the prefix sum over the entire array.

This adjustment involves adding the cumulative sum of elements preceding the current segment to the prefix sums gathered. This step ensures that each element in the final array is the correct prefix sum, taking into account all previously processed segments. Finally, the root process prints the adjusted prefix sums in a formatted manner.

Q4

Main Idea :-

As we need to take input from one spawned process (i.e here process with rank 0) , we broadcasted the n value to all the processes and the matrix is divided into submatrices (rows) and scattered across the available processes. Each process receives a portion of the matrix, depending on the number of rows it needs to handle (`rows_per_process - n/p`) + 1 (if rank < n%p as there will be some rows left which is not divisible by p i.e less than p).

- **Pivot Row Selection**: The process that owns the current pivot row (the row currently being used for elimination) normalizes it by dividing all elements in the row by the pivot element.
- **Broadcasting the Pivot Row**: The normalized pivot row is then broadcasted to all other processes using `MPI_Bcast`.
- **Elimination**: Each process then uses the broadcasted pivot row to eliminate the corresponding column in its local rows, ensuring that all rows below the pivot row have zeros in the pivot column.

The code uses `MPI_Barrier` before and after broadcasting the pivot row to ensure that all processes synchronize and work with the correct pivot row. This is important because `MPI_Bcast`. needs to be done only after the owner process updates the pivot_row_A and pivot_row_AI.

The final inverse matrix is gathered back to the root process using `MPI_Gatherv`. The root process then prints the inverted matrix.

Q5

Main Idea :-

As we need to take input from one spawned process (i.e here process with rank 0) , we broadcast the size `n` to all processes using `MPI_Bcast`.

Next, the entire array is also broadcasted to all processes, so every process has the full data. Then, a 2D dynamic programming table `dp`(where dp[i][j] (i<=j) refers to the value of minimum number of computations to compute value of matrix multiplications of matrices from index i to j), and a diagonal vector `diag` are initialized, which will be used for computing the solution to the problem. The main computation involves iterating over different lengths (`len`) of the segments to be processed.

For each length, if the length is greater than 2, we resize the `diag` vector to hold the values of the next diagonal in the DP table, and the values of the previous diagonal are copied into the DP table. Here diagonal refers to all the straight lines joining dp[0][i] to dp[n-i][n], for every i from 0 to n.

The work for each segment/diagonal length is distributed among the processes. Each process calculates the cost for its assigned range of matrix segments. The range is computed based on the number of queries each process should handle (`queries_per_process`) and any remainder (`remainder`) that accounts for uneven distribution.

Each process then computes its local results using the DP table results from the previous diagonals, and stores them in `local_arr`. After computation, the local results are gathered back to the root process using `MPI_Gatherv`. The root process collects these results in the `diag` vector, ensuring that each process's computed data is placed correctly in the global context.

The root process then broadcasts the updated diagonal vector to all other processes so that they can use it for further calculations in the next iteration. This cycle continues until all segments have been processed.

Finally, once all the computations are done, the root process outputs the time taken for the computation and the result stored in `diag[0]`(This is basically dp[0][n] or the minimum number of computations to compute multiplication of matrices from index 0 to n).