MPI Distributed Systems Assignment Report

Question 1: Program for Finding K Nearest Neighbours

The program follows these steps:

1. **Initialization:**
    - The program initializes MPI, determines the rank of each process, and the total number of processes.
2. **Input Handling:**
    - The root process (rank 0) reads input values: the number of points in sets `P` (`N`) and `Q` (`M`), and the value of `K`.
    - Points in sets `P` and `Q` are also read by the root process.
    - The values of `N`, `M`, and `K` are broadcast to all other processes using `MPI_Bcast`.
3. **Data Distribution:**
    - The points in set `Q` are divided among the processes. Each process receives a chunk of points (approximately `M/size` points, with some processes possibly getting one extra point if `M` is not divisible by the number of processes).
    - The root process sends the corresponding subset of `Q` to each process using `MPI_Scatterv`.
4. **Local Computation:**
    - Each process computes the `K` closest points in `P` for each point in its assigned subset of `Q`.
    - For each point in the local subset of `Q`, the program calculates the Euclidean distance to all points in `P`.
    - A max-heap is used to maintain the closest `K` points efficiently.
    - The result for each point in `Q` is stored locally.
5. **Gathering Results:**
    - Each process sends its local results back to the root process using `MPI_Gatherv`.
    - The root process collects all results and prints them.
6. **Output:**
    - The root process prints the `K` closest points for each point in `Q` in order.

Time Complexity

1. **Broadcasting `P` and `Q`:**

- Broadcasting `P`: `O(N)` per process for receiving.
- Broadcasting `Q` for scatter: `O(M/size)` per process for receiving.

2. **Local Computation:**

   - Each process handles `M/size` points from `Q`. For each point in `Q`, calculating the distance to all `N` points in `P` takes `O(N)`.
   - Maintaining a max-heap of size `K` for the closest points takes `O(log K)` per insertion.
   - Total time per process: `O((M/size) * N * log K)`.

3. **Gathering Results:**

   - Gathering results from all processes has a cost of `O(M)` in total for the root process.

   - **Total Time Complexity:** `O(M * N * log K / size)`.

Space Complexity

1. **Memory for Point Storage:**

   - The entire `P` set of `N` points is stored on all processes: `O(N)`.
   - Each process stores approximately `M/size` points from `Q`: `O(M/size)`.

2. **Local Results:**

   - Each process stores results for its local subset of `Q`, consuming `O(K * M/size)` space.

3. **Global Results:**

   - The root process stores all results, requiring `O(K * M)` space.

   - **Total Space Complexity:**

     - Per process: `O(N + K * M/size)`.
     - On the root process (rank 0): `O(N + K * M)`.

Question 2: MPI Program for Julia Set Calculation

The program performs the following steps:

1. **Initialization:**

   - MPI is initialized, and each process retrieves its rank and the total number of processes (`size`).

2. **Input Handling:**

   - The root process (rank 0) reads input values, including the dimensions of the grid (`N`, `M`), the maximum number of iterations (`K`), and the constant complex number `c = c_real + c_img*i` used in the Julia set formula.

- These values are broadcasted to all processes using `MPI_Bcast`.

3. **Grid Partitioning:**

   - The grid is partitioned row-wise among the processes. Each process is responsible for computing the Julia set values for a subset of rows. The number of rows each process handles is determined by `rows_per_proc`, with a remainder handled by some processes to ensure all rows are covered.

4. **Local Computation:**

   - Each process computes the Julia set for its assigned rows.
   - The grid points are mapped to complex numbers, `z0 = x + yi`, within the range `[-1.5, 1.5]` in both real and imaginary parts.
   - For each grid point, the Julia set membership is determined by iterating the complex function `z = z * z + c` up to `K` times. If the magnitude of `z` exceeds a threshold `T`, the point is not part of the Julia set.

5. **Gathering Results:**

   - Each process sends its computed results to the root process using `MPI_Gatherv`. This function gathers the computed data from all processes and assembles them into the final result.
   - The root process collects the results into a global grid that represents the entire Julia set.

6. **Output:**

   - The root process prints the results, where each grid point is represented by either `1` (indicating the point is within the Julia set) or `0` (indicating the point is outside the set).

Time Complexity

1. **Broadcasting Input:**

   - Broadcasting `N`, `M`, `K`, and the complex number `c` takes `O(1)` time for the root process, but all processes need to receive this data, making it `O(1)` per process.

2. **Local Computation:**

   - Each process computes the Julia set for approximately `N/size` rows. For each point in the grid, computing the Julia set takes up to `K` iterations, resulting in:
   - **Time Complexity per process:** `O((N/size) * M * K)`.

3. **Gathering Results:**

   - Gathering the computed results at the root process has a cost of `O(N * M)` for the root process as it must collect data from all processes.

   - **Total Time Complexity:** The overall time complexity is dominated by the local computation, leading to:

     - `O(N * M * K / size)`.

Space Complexity

1. **Memory for Grid Storage:**
   - Each process stores approximately `N/size` rows of the grid, consuming `O((N/size) * M)` space.
2. **Global Result Storage:**
   - The root process stores the entire grid, which requires `O(N * M)` space.
3. **Communication Buffers:**
   - Each process maintains buffers for sending and receiving data, which add an additional `O(M)` space for the row data.
   - **Total Space Complexity:**
     - Per process: `O((N/size) * M)`.
     - On the root process (rank 0): `O(N * M)` for the final output grid.

Question 3: Program for Finding the Inverse of a Matrix

The program follows these steps:

1. **Initialization:**
   - The program initializes MPI, determines the rank of each process, and the total number of processes.
2. **Input Handling:**
   - The root process (rank 0) reads the size of the matrix (N) and the matrix (A) itself.
   - The value of (N) is broadcast to all other processes using `MPI_Bcast`.
3. **Data Distribution:**
   - The matrix (A) is divided into rows, and each process receives a portion of these rows. The number of rows assigned to each process is approximately (N/\text), with some processes possibly receiving an additional row if (N) is not divisible by the number of processes.
   - The rows of the matrix (A) are scattered to each process using `MPI_Scatterv`. Each process also augments its portion of the matrix with the corresponding identity matrix rows.
4. **Local Computation:**
   - Each process performs Gaussian elimination on its local rows, normalizing the pivot row and broadcasting it to all other processes.
   - Each process uses the broadcasted pivot row to eliminate the corresponding elements in its own rows, ensuring the matrix is reduced to row-echelon form.
5. **Gathering Results:**

- Once all rows are reduced, each process sends its portion of the inverse matrix back to the root process using `MPI_Gatherv`.
- The root process collects all the portions to reconstruct the complete inverse matrix.

6. **Output:**

   - The root process prints the inverse matrix with exactly two decimal places for each element.

Time Complexity

1. **Broadcasting (N) and Scattering (A):**

   - Broadcasting (N): $O(1)$ per process.
   - Scattering (A): $O(N^2/\text{size})$ per process for receiving.

2. **Gaussian Elimination:**

   - Each process handles approximately (N/\text) rows. The time to normalize the pivot row and eliminate elements across all rows is dominated by the row-wise operations.
   - **Overall time complexity for Gaussian elimination:** $O(N^3/\text{size})$.

3. **Gathering Results:**

   - Gathering the inverse matrix from all processes: $O(N^2)$ for the root process.

   - **Total Time Complexity:** $O(N^3/\text{size})$.

Space Complexity

1. **Memory for Matrix Storage:**

   - Each process stores its portion of (A), which is approximately $O(N^2/\text{size})$.
   - Each process also needs to store the identity matrix portion for augmentation: $O(N^2/\text{size})$.

2. **Local Results:**

   - The inverse matrix's portion: $O(N^2/\text{size})$ per process.

3. **Global Results:**

   - The root process stores the entire inverse matrix: $O(N^2)$.

   - **Total Space Complexity:**

     - Per process: $O(N^2/\text{size})$.
     - On the root process (rank 0): $O(N^2)$.

Question 4: Program for Parallel Matrix Chain Multiplication Problem

The program follows these steps:

1. **Initialization:**
   - The program initializes MPI, determines the rank of each process, and the total number of processes.
2. **Input Handling:**
   - The root process (rank 0) reads the number of matrices (N) and their dimensions from the input.
   - The dimensions array is broadcast to all other processes using `MPI_Bcast`.
3. **Data Distribution:**
   - The dynamic programming table for matrix chain multiplication is computed in parallel. The subproblems are distributed across the processes, with each process handling a subset of the table.
   - Each process computes the cost of multiplying a specific range of matrices, based on its rank.
4. **Local Computation:**
   - Each process calculates the minimum number of scalar multiplications required for its assigned portion of the matrix chain.
   - The results from each step are synchronized using `MPI_Barrier` and broadcast to all processes to ensure consistency.
5. **Gathering Results:**
   - The final minimum multiplication cost is gathered from all processes and collected by the root process.
   - The root process then prints the result.

Time Complexity

1. **Broadcasting Dimensions:**
   - Broadcasting the dimensions array: $O(N)$ per process.
2. **Dynamic Programming Table Calculation:**
   - Each process handles a portion of the dynamic programming table, with the computation taking $O(N^3/\text{size})$ per process.
3. **Synchronization and Broadcasting Results:**
   - Synchronization overhead due to barriers: $O(N^2 \log \text{size})$ overall.
   - **Total Time Complexity:** $O(N^3/\text{size})$.

Space Complexity

1. **Memory for Storing Dimensions:**
   - Each process stores the dimensions array: $O(N)$.
2. **Dynamic Programming Table:**
   - Each process maintains a portion of the table: $O(N^2/\text{size})$.
3. **Global Results:**
   - The root process stores the entire table for final output: $O(N^2)$.
   - **Total Space Complexity:**
     - Per process: $O(N^2/\text{size})$.
     - On the root process (rank 0): $O(N^2)$.

Question 5: MPI Program for Parallel Prefix Sum Calculation

The program performs the following steps:

1. **Initialization:**
   - MPI is initialized, and each process retrieves its rank (`rank`) and the total number of processes (`size`).
2. **Input Handling:**
   - The root process (rank 0) reads the input data, including the size of the sequence `n` and the sequence of `n` double values.
   - These values are broadcasted to all processes using `MPI_Bcast`.
3. **Grid Partitioning:**
   - The sequence is divided into `size` parts, one for each process.
   - The division accounts for the possibility that `n` may not be perfectly divisible by `size`, so some processes may handle one additional element (`remainder`).
4. **Local Computation:**
   - Each process computes the prefix sum for its assigned portion of the sequence.
   - For each element in its local array, the prefix sum is calculated iteratively by adding the current element to the sum of all previous elements in the local array.
5. **Gathering Local Sums:**
   - Each process calculates the last element of its local prefix sum and sends it to the root process using `MPI_Gather`.

- The root process uses these last elements to compute offsets for each process, which are then broadcasted back to all processes using `MPI_Bcast`.

6. **Adjusting Local Prefix Sums:**

   - Each process adjusts its local prefix sum by adding the offset it received. This ensures that the prefix sum is correctly accumulated across all processes.

7. **Gathering Results:**

   - The adjusted local prefix sums from all processes are gathered at the root process using `MPI_Gatherv`.
   - The root process assembles the final prefix sum sequence and prints it.

Time Complexity

1. **Broadcasting Input:**

   - Broadcasting the input size and data takes `O(1)` time for the root process and `O(1)` for each receiving process.

2. **Local Prefix Sum Calculation:**

   - Each process computes the prefix sum for its portion of the sequence.
   - **Time Complexity per process:** `O(n/size)`.

3. **Gathering Last Elements:**

   - The root process gathers the last elements from all processes, which takes `O(size)` time for the root process.

4. **Broadcasting Offsets:**

   - The offsets are broadcasted to all processes, which takes `O(size)` time.

5. **Adjusting and Gathering Results:**

   - Adjusting local prefix sums and gathering the final results take `O(n)` time for the root process.

   - **Total Time Complexity:** The total time complexity is `O(n/size)`, dominated by the local prefix sum calculation.

Space Complexity

1. **Memory for Local Arrays:**

   - Each process stores its portion of the sequence, which requires `O(n/size)` space.

2. **Global Result Storage:**

   - The root process stores the entire prefix sum sequence, requiring `O(n)` space.

3. **Communication Buffers:**

   - Additional space is used for communication buffers, such as `sendcounts`, `displacements`, and `offsets`, which add `O(size)` space.

   - **Total Space Complexity:**

     - Per process: `O(n/size)`.
     - On the root process (rank 0): `O(n)` for the final output sequence.