Question-4: Inverse of a Matrix (24 points)

Solution Approach

Was told to use the Row Reduction Method (Gaussian Inversion -¿ Identity Matrix Augmentation) in the document, for Matrix Inversion.

1. Preprocess the matrix. If the ith element in ith row is 0, find a row somewhere below it where this element is not 0, and swap with it.
2. Split up chunk of rows and spread the tasks out across the processors. Each processor has to process a certain number of rows depending on the number of elements in the matrix. Note that some processors may have to process at most 1 row less than the others.
3. While making identity matrix as a whole, you just need to make one for the current row only. Put a 1 in the (i+1)th position (1-indexed) for row i (where i is the index of the row in the ORIGINAL matrix), and the rest are zeros.
4. Now, iterating over the rows, pivot row i, and get the process that owns this row to normalize it and broadcast just this 1 row to all the other processes so that they can subtract it from their own rows (except pivot row) after multiplying it with some factor.
5. Combine the rows of the initial identity matrix (augmented matrix) which is now A^{-1} itself into root process, and return the answer.

Highlights of Program

If N is not divisible by p, then the first few processes take up 1 extra row until all the remaining rows are used up. Thus, every process computes matrix inversion of at most 1 row more than others. This is a nearly equal distribution of data amongst different processors.

Furthermore, in my testing, `MPI_Scatterv` was found to be *severely* bad, and thus, since I had already used `MPI_Scatter` in Question-3 and demonstrated its inefficacy, I have decided to go for the alternative approach instead, and instead, broadcast the ENTIRE matrix to all the elements so that they can store it with themselves, and create a smaller `local_row` vector, which contains only the rows that they need to compute. This way, I save time on scattering, as broadcast is just so much faster.

Total Time Complexity of Approach

O(N^3 / p), since every process has to subtract N rows from each of its N/p rows of N elements each.

Total Message Complexity of Approach

For broadcasting number of elements, it is O(log p) using `MPI_Bcast`.

For broadcasting initial data, it is O(3 * log p) using `MPI_Bcast`, since I broadcast a matrix of size n*n, a row_swap_tracker of size n and a row_owner_tracker of size n. So, N^2 * sizeof(float) + 2*N * sizeof(int) data is received by each process.

For broadcasting pivot row in each row iteration, it is O(N * log p) using `MPI_Bcast`, as there are N broadcasts made in total.

For gathering all the final rows, there are N rows with N elements each and thus, using `MPI_Gatherv`, it comes out to be O(N^2 * log p).

All things considered, it comes out to be **O(log p + N * log p + N^2 * log p)**.

Space Requirements of Solution

Since each process has to store the entire matrix, it is O(N^2) for the matrix per processor. Additionally, there are a couple of O(N/p * N) row vectors, and O(N) vectors to determine row ownership, and O(p) vectors to determine displacements (offsets).

Overall, it comes out to be O(N^2 + N^2/p + N + p).

Performance Scaling from 1 to 12 processes

Size of testcase: *1000x1000 matrix*

Number of Processors	Time Elapsed
1	7.6162s
2	6.5852s
3	4.6765s
4	3.63247s
5	2.86221s
6	2.44543s
7	2.28356s
8	1.86037s
9	1.65244s
10	1.48994s
11	1.41684s
12	1.29692s