Grid Code:

The grid code implements matrix multiplication in a parallel computing environment using MPI It sets up a 2D Cartesian grid of processes, where each process is responsible for computing a part of the resultant matrix. The steps below describe how the program works, including specific MPI calls used at each stage.

Step 1: Initialization

- **MPI_Init**: Initializes the MPI environment. This must be called before any other MPI functions.
- **MPI_Comm_rank**: Determines the rank (unique identifier) of the current process in the MPI_COMM_WORLD communicator, which includes all processes started by MPI.
- **MPI_Comm_size**: Determines the total number of processes in the MPI_COMM_WORLD communicator.

Step 2: Setup Process Grid

- **findMultiples**: A custom function (not an MPI call) that finds two factors of the total number of processes that are as close as possible to each other. These factors are used to define the dimensions of a 2D grid of processes.
- **MPI_Cart_create**: Creates a new communicator (grid_comm) for a Cartesian topology and assigns it to the processes. The processes are logically arranged in a 2D grid based on the dimensions calculated by findMultiples.

Step 3: Distribute Work

- Each process calculates its coordinates within the 2D grid using **MPI_Cart_coords**, which is essential for determining the portion of the matrices each process will handle.
- The matrices B and C are broadcasted to all processes using **MPI_Bcast**. The root process (rank 0) sends the entire matrices, and all other processes receive them.

Step 4: Local Computation

- Each process computes a sub-section of the resultant matrix A. This is done by performing matrix multiplication on the locally stored sections of matrices B and C.

Step 5: Gather Results

- The root process collects the computed sub-matrices from all processes. This is done in two steps:
  - The root process directly copies its part of the computed matrix.
  - For other processes, **MPI_Recv** is used by the root process to receive their computed sub-sections, which are then placed correctly in the final matrix A.
- Non-root processes use **MPI_Send** to send their computed sub-matrices to the root process.

Step 6: Output Result

- The root process writes the final matrix A to a file. This involves standard file I/O operations in C and doesn't use MPI calls.

Step 7: Cleanup

- Memory allocated for matrices and MPI resources are freed up. This includes freeing dynamically allocated 2D arrays for matrices A, B, C, and the local sub-matrix APart.
- **MPI_Finalize**: Terminates the MPI environment. No MPI calls may be made after this point.

Ring Code:

The ring program implements a parallel matrix multiplication using MPI in a ring topology. It follows a series of steps, including setting up the MPI environment, distributing the workload among processes arranged in a ring, performing local computations, and gathering the results. Below is a step-by-step explanation of the program:

Step 1: Initialization

- **MPI_Init**: Initializes the MPI environment, preparing it for MPI operations.
- **MPI_Comm_rank & MPI_Comm_size**: These functions determine the rank (unique identifier) of the current process and the total number of processes in the MPI_COMM_WORLD communicator, respectively.

Step 2: Setup Ring Topology

- **MPI_Cart_create**: Creates a new communicator (ring_comm) with a ring topology. This function configures the processes in a logical ring, allowing for cyclic communication patterns.
- **MPI_Cart_coords & MPI_Cart_rank**: Determine the new coordinates of each process in the ring topology and reassess new ranks , ensuring each process knows its position in the ring.

Step 3: Determine Work Distribution

- Each process calculates how many rows of the matrices it will handle, dividing the total number of rows (N) by the number of processes. The root process handles any remaining rows (in case N is not perfectly divisible).
- **MPI_Cart_shift**: Identifies source and destination ranks for each process in the ring, facilitating the send/receive operations between neighbours.

Step 4: Data Distribution

- The root process initializes matrices B and C with user-provided values. Then, it directly copies the extra rows (if any) into its local sub-matrix (BPart).

- **MPI_Scatter**: Distributes contiguous segments (rows) of matrix B from the root process to all processes in the ring, including the root itself.

- **MPI_Bcast**: Broadcasts the entire matrix C to all processes, as each process needs the complete matrix C for local computations.

Step 5: Local Computation

- Each process computes its assigned portion of the matrix multiplication result. This involves iterating over its local rows of matrix B (or BPart) and the entire matrix C, performing dot products to compute corresponding elements of the result matrix A (or APart).

Step 6: Gather Results

- **MPI_Gather**: Collects the computed sub-matrices from all processes back to the root process. The root process first handles its extra rows (if any) and then gathers results from other processes starting from the appropriate offset.

Step 7: Output and Cleanup

- The root process writes the final result matrix A to a file, providing a tangible output of the matrix multiplication.

- All dynamically allocated memory for matrices and their sub-parts is freed to prevent memory leaks.

- **MPI_Finalize**: Cleans up the MPI environment, terminating the MPI execution environment

Compilation and Execution:

1. Compiling the Ring Topology Program:

```
mpicc matrix_matrix_ring.c -o mmr -DN=64 -lm
```

- **mpicc**: This is the MPI compiler wrapper for C programs. It includes the necessary MPI libraries during the compilation process.
- **matrix_matrix_ring.c**: The source file for the program that implements matrix multiplication using a ring topology.
- **-o mmr**: Specifies the output executable's name. In this case, mmr is the name of the executable file generated by the compiler.
- **-DN=64**: This part of the command defines a preprocessor macro N with the value 64. It's equivalent to adding #define N 64 at the top of your source file. The macro N represents the size of the matrices to be used in the matrix multiplication.
- **-lm**: Links the math library (libm). This is necessary if the program uses mathematical functions.

2. Compiling the Grid Topology Program:

```
mpicc matrix_matrix_grid.c -o mmg -DN=64 -lm
```

- **mpicc**: MPI compiler wrapper for C programs.
- **matrix_matrix_grid.c**: The source file for the program that implements matrix multiplication using a grid topology.
- **-o mmg**: Specifies that the output file (executable) should be named mmg.
- **-DN=64**: Defines a preprocessor macro N with the value 64, setting the size of the matrices for the matrix multiplication.
- **-lm**: Links the math library, which may be required for mathematical operations within the program.

Execution Command for Ring:

```
mpirun -np 11 ./mmr < input.txt
```

- **mpirun**: This is the command used to run MPI programs. It starts multiple processes of the specified MPI program.
- **-np 11**: Specifies the number of processes to start. In this case, 11 processes will be created.
- **./mmr**: Indicates the executable to run, which in this case is the mmr file generated by compiling matrix_matrix_ring.c.
- **< input.txt**: Redirects the standard input to read from a file named input.txt. This file contains the input data the program requires, such as the elements of the matrices to be multiplied.

Execution Command for Grid:

Same as above but replace mmr with mmg.