Parallel Programming MPI Part 1 - **Cannon Algorithm Task A**

Pavithra Purushothaman (fd0001571)
Matrikel Nummer: 1535949

In my implementation of **Cannon's algorithm**, I focused on efficiently distributing and exchanging matrix elements among processes. I started by having Rank 0 read the input matrix size N and matrices A and B from text files generated by the "cannon_input_generator.c" file. To distribute these matrices, I used a combination of **point-to-point** and **collective communication** methods. For broadcasting the matrix size N to all processes, I used the MPI_Bcast operation.

For the initial distribution, I chose to use MPI_Send from Rank 0 to send blocks of the matrices to all other processes. I created a custom MPI datatype (block_type) to efficiently send these blocks. The receiving processes used MPI_Recv to get their respective blocks. This approach allowed me to distribute the matrices proportionally, with each process receiving one block from each matrix.

To set up the process grid, I used MPI_Cart_create to establish a **2D Cartesian topology**. This topology is beneficial because it organizes processes in a grid, facilitating the systematic shifting of matrix blocks during the algorithm's execution.

For the exchange of matrix elements during the algorithm, I implemented a system using MPI_Sendrecv_replace. This function allowed me to simultaneously send and receive data, which was perfect for the shifting operations in Cannon's algorithm. I used it for both the initial alignment of matrices and the repeated shifts during the main computation loop.

```
// Shifting matrix A left by one
MPI_Cart_shift(cart_comm, 1, -1, &shift_src, &shift_dst);
MPI_Sendrecv_replace(local_A, block_size * block_size, MPI_INT, shift_dst, 0, shift_src, 0, cart_comm, &status);

// Shifting matrix B up by one
MPI_Cart_shift(cart_comm, 0, -1, &shift_src, &shift_dst);
MPI_Sendrecv_replace(local_B, block_size * block_size, MPI_INT, shift_dst, 0, shift_src, 0, cart_comm, &status);
```

I chose MPI_Sendrecv_replace over separate send and receive operations because it simplified the code and reduced the risk of deadlocks. It's a blocking operation, which ensured that each step of the algorithm was completed before moving to the next, maintaining the correct sequence of operations.

At the end of the computation, I used MPI_Gather to collect the results from all processes back to Rank 0. This collective operation was ideal for assembling the final result matrix C from the distributed blocks.

Throughout the implementation, I primarily used **blocking communications**. While non-blocking communications can offer performance benefits in some scenarios, I found that the simplicity and reliability of blocking communications were well-suited for this implementation of Cannon's algorithm. Future improvements could involve exploring non-blocking communications or other optimizations to further enhance performance.

References:

1. Used lecture slides for understanding different communication mechanisms.
2. Tutorial on Cannon's algorithm for understanding, here.