# Openmp

The OpenMP directives used in this code are:

1. #pragma omp parallel for collapse(2) num\_threads(p): This directive creates a parallel region with p threads and parallelizes the nested loops for initializing matrices A and B. The collapse(2) clause is used to collapse the two levels of loops into a single loop to improve parallelization.
2. #pragma omp master: This directive is used to specify that the following code block should only be executed by the master thread (thread ID 0).
3. #pragma omp parallel num\_threads(p): This directive creates a parallel region with p threads for the main matrix multiplication computation.
4. #pragma omp barrier: This directive is used to synchronize all threads at certain points in the code, ensuring that they all reach the barrier before continuing execution.
5. #pragma omp master: This directive is used again to specify that certain code blocks should only be executed by the master thread, like recording the start and end times, rotating the blocks of matrices A and B, and printing the output matrix C.

## Steps

1. Each thread computes its row and column position in a 2x2 grid:
   * Thread 0: row = 0, col = 0
   * Thread 1: row = 0, col = 1
   * Thread 2: row = 1, col = 0
   * Thread 3: row = 1, col = 1

Matrix A and B are divided into 2x2 blocks:

For example,

Matrix A (in blocks):

┌─────┬─────┐

│ 1 2 │ 1 2 │

│ 2 1 │ 2 1 │

├─────┼─────┤

│ 2 1 │ 2 1 │

│ 1 2 │ 1 2 │

└─────┴─────┘

Matrix B (in blocks):

┌─────┬─────┐

│ 1 1 │ 2 2 │

│ 2 2 │ 1 1 │

├─────┼─────┤

│ 2 2 │ 1 1 │

│ 1 1 │ 2 2 │

└─────┴─────┘

1. The main loop iterates over sqrt\_p (2 iterations in this case):
   * Iteration 0:
     + Each thread computes the product of the corresponding blocks of A and B and adds the result to the corresponding block of C.
     + Thread 0: C[0][0] block = A[0][0] block \* B[0][0] block
     + Thread 1: C[0][1] block = A[0][0] block \* B[0][1] block
     + Thread 2: C[1][0] block = A[1][0] block \* B[0][0] block
     + Thread 3: C[1][1] block = A[1][0] block \* B[0][1] block

*Like normal matrices multiplication.*

* + After computing, the threads synchronize at the barrier.
  + The master thread rotates the blocks of matrices A and B:
    - Matrix A: Rotate each row to the left by 1 position
    - Matrix B: Rotate each column up by 1 position

*Rotate the blocks to align necessary blocks of A,B for the next iteration.*

* + The threads synchronize again at the barrier.
  + Iteration 1:
    - Each thread computes the product of the corresponding blocks of A and B and adds the result to the corresponding block of C.
    - Thread 0: C[0][0] block += A[0][1] block \* B[1][0] block
    - Thread 1: C[0][1] block += A[0][1] block \* B[1][1] block
    - Thread 2: C[1][0] block += A[1][1] block \* B[1][0] block
    - Thread 3: C[1][1] block += A[1][1] block \* B[1][1] block
  + After computing, the threads synchronize at the barrier.

1. The master thread prints the resultant matrix C.
2. Finally, the total time taken for the computation is calculated and printed.

# MPI

In this code, the following MPI functions are used:

1. MPI\_Init: Initializes the MPI environment. It must be called before any other MPI functions are used.
2. MPI\_Comm\_rank: Determines the rank (unique process ID) of the calling process within the specified communicator (MPI\_COMM\_WORLD in this case).
3. MPI\_Comm\_size: Determines the total number of processes within the specified communicator (MPI\_COMM\_WORLD in this case).
4. MPI\_Cart\_create: Creates a new communicator with a Cartesian topology based on the input dimensions, periods, and reorder flag. In this code, it creates a 2D grid of processes with dimensions n by n, where n is the square root of the total number of processes.
5. MPI\_Cart\_coords: Determines the Cartesian coordinates of a process within a Cartesian topology communicator. In this code, it returns the row and column coordinates of the current process within the 2D grid.
6. MPI\_Cart\_shift: Computes the rank of the source and destination processes along a specified axis after a specified shift. In this code, it is used to perform the initial alignment of blocks and to shift the blocks of matrices A and B during the Cannon's algorithm iterations.
7. MPI\_Sendrecv\_replace: Sends and receives a message using a single buffer. In this code, it is used to exchange blocks of matrices A and B between processes during the initial alignment and the shifts in the Cannon's algorithm iterations.
8. MPI\_Wtime: Returns the elapsed time in seconds since an arbitrary starting point. In this code, it is used to measure the time taken for the local block multiplications.
9. MPI\_Reduce: Applies a reduction operation on the data from all processes within a communicator and returns the final result to the root process. In this code, it is used to calculate the maximum elapsed time across all processes to measure the performance of the algorithm.
10. MPI\_Barrier: Synchronizes all processes within a communicator. In this code, it ensures that all processes have completed their work before the output is printed.
11. MPI\_Gather: Gathers data from all processes within a communicator and combines the data into a single buffer on the root process. In this code, it is used to gather the blocks of matrices A, B, and C into gathered\_A, gathered\_B, and gathered\_C, respectively, on the root process (rank 0).
12. MPI\_Finalize: Cleans up the MPI environment and should be called before the program exits.

## Steps

1. Initialization: The code starts by initializing the MPI environment, obtaining the rank and size of the MPI communicator, and checking if the number of processes is a perfect square. If it is not, an error message is displayed, and the program terminates.
   * the number of processes is checked for being a perfect square to ensure that the processes can be arranged in a 2D Cartesian topology with an equal number of rows and columns.
2. Matrix setup: The code calculates the dimensions of the matrices (A, B, and C) that will be multiplied in parallel. It also allocates memory for these matrices and populates matrices A and B with random integers. Matrix C is initialized with zeros.
3. Cartesian topology creation: The code creates a Cartesian topology for the MPI processes and assigns each process a set of coordinates within this topology.
4. Initial alignment of blocks: The matrices A and B are divided into blocks, and each block is shifted according to the Cartesian coordinates of the process. This is done using the MPI\_Sendrecv\_replace function, which sends and receives data in a single call.
5. Block multiplication: The code starts measuring the time taken for the matrix multiplication. Each process performs the multiplication of its local blocks of matrices A and B, updating its local block of matrix C.
6. Block shifting and accumulation: The code iterates n - 1 times, where n is the square root of the number of processes. In each iteration, the blocks of matrices A and B are shifted along their respective axes in the Cartesian topology. The local block multiplication is performed again, and the results are accumulated in the local block of matrix C.

why we use (int l = 1; l < n; l++):

n represents the number of blocks along one dimension of the square grid in the 2D Cartesian topology,

The algorithm has already performed the first block-wise multiplication and the initial alignment of matrix A and matrix B blocks before entering this loop. Therefore, it starts with l = 1 to perform the remaining n - 1 block-wise multiplications.

In each iteration of the loop, the blocks of matrix A and matrix B are shifted, and the partial results are accumulated in matrix C. By iterating n - 1 times, each process will perform block-wise multiplication with every possible pair of A and B blocks that contribute to the final result.

1. Time measurement: After all iterations are complete, the code measures the time taken for the matrix multiplication and calculates the maximum elapsed time across all processes.
2. Synchronization and data gathering: The code synchronizes all processes using MPI\_Barrier. Then, it gathers the blocks of matrices A, B, and C from all processes and stores them in the root process (rank 0).
3. Printing results: If the current process is the root process (rank 0), it prints the complete matrices A, B, and C using the print\_complete\_matrix function. The function iterates through the blocks of the matrices and prints them in a formatted manner. Finally, the maximum elapsed time for the matrix multiplication is printed.
4. Cleanup: The code deallocates the memory used for matrices A, B, and C, as well as the gathered matrices in the root process. The MPI environment is finalized, and the program exits.

The Cartesian topology is a convenient way to organize and manage the MPI processes in a grid-like structure for parallel matrix multiplication. It simplifies the process of shifting and accumulating the blocks of matrices A and B, making the parallel algorithm more efficient.

The Cartesian topology is created using the MPI\_Cart\_create function, which takes the following parameters:

* The input communicator, typically MPI\_COMM\_WORLD.
* The number of dimensions (2 in this case, as we have a 2D grid structure).
* An array containing the dimensions of the grid (e.g., {n, n} where n is the square root of the number of processes).
* An array specifying whether the grid is periodic along each dimension (e.g., {1, 1} for periodic in both dimensions, meaning that the edges wrap around).
* A flag to allow or disallow reordering of the ranks (0 to disallow reordering in this code).
* A pointer to the new communicator with the Cartesian topology (in this case, &comm\_cart).

It is used in the following ways:

1. Assigning coordinates: The MPI\_Cart\_coords function is used to obtain the Cartesian coordinates (row, column) of a process within the topology. These coordinates are used in the initial alignment of the blocks of matrices A and B.
2. Initial alignment of blocks: The code shifts the blocks of matrices A and B according to the Cartesian coordinates of the process. This is done using the MPI\_Cart\_shift function, which takes the following parameters:
   * The Cartesian communicator (comm\_cart).
   * The direction of the shift (0 for rows, 1 for columns).
   * The size of the shift (equal to the row or column coordinate, depending on the direction).
   * Pointers to the ranks of the source and destination processes for the data transfer.

The MPI\_Sendrecv\_replace function is then used to send and receive the blocks of matrices A and B, completing the initial alignment.

Block shifting and accumulation: In each iteration of the loop, the blocks of matrices A and B are shifted along their respective axes in the Cartesian topology:

Calculate the shift amount for matrix A blocks and matrix B blocks using the process coordinates.

* For matrix A, the blocks are shifted to the left by an amount equal to the row coordinate of the process.
* For matrix B, the blocks are shifted upwards by an amount equal to the column coordinate of the process.

Again, the MPI\_Cart\_shift and MPI\_Sendrecv\_replace functions are used for this purpose. **After the shifting, the local block multiplication is performed again, and the results are accumulated in the local block of matrix C.**

Visualized explanation

Matrix A:

1 2 1 2

2 1 2 1

2 1 2 1

1 2 1 2

Matrix B:

1 1 2 2

2 2 1 1

2 2 1 1

1 1 2 2

1. First, create a Cartesian topology with 2 rows and 2 columns. Assign coordinates to each process within the topology:

P0 (0,0) | P1 (0,1)

---------+---------

P2 (1,0) | P3 (1,1)

1. Divide the matrices A and B into blocks:

Matrix A blocks:

[1 2] [1 2]

[2 1] [2 1]

[2 1] [2 1]

[1 2] [1 2]

Matrix B blocks:

[1 1] [2 2]

[2 2] [1 1]

[2 2] [1 1]

[1 1] [2 2]

1. Perform initial alignment of the blocks **according to the process coordinates**:

The initial alignment is performed as follows:

1. For each process, obtain its Cartesian coordinates (row, column) within the topology.
2. Calculate the shift amount for matrix A blocks and matrix B blocks using the process coordinates.
   * For matrix A, the blocks are shifted to the left by an amount equal to the row coordinate of the process.
   * For matrix B, the blocks are shifted upwards by an amount equal to the column coordinate of the process.

P0 (0,0): No shift for matrix A blocks, no shift for matrix B blocks.

P1 (0,1): Shift matrix A blocks to the left by 0, shift matrix B blocks upwards by 1.

P2 (1,0): Shift matrix A blocks to the left by 1, shift matrix B blocks upwards by 0.

P3 (1,1): Shift matrix A blocks to the left by 1, shift matrix B blocks upwards by 1.

Matrix A blocks after initial alignment:

[1 2] [1 2]

[2 1] [2 1]

[1 2] [1 2]

[2 1] [2 1]

Matrix B blocks after initial alignment:

[1 1] [2 2]

[2 2] [1 1]

[1 1] [2 2]

[2 2] [1 1]

1. Perform local block multiplication in each process:

P0: [1 2] \* [1 1] = [5 5]

[2 1] [2 2] [4 4]

P1: [1 2] \* [1 1] = [5 5]

[2 1] [2 2] [4 4]

P2: [1 2] \* [2 2] = [4 4]

[2 1] [1 1] [5 5]

P3: [1 2] \* [2 2] = [4 4]

[2 1] [1 1] [5 5]

1. Perform block shifting and accumulation:

Iteration 1:

* Shift matrix A blocks 1 step to the left.
* Shift matrix B blocks 1 step up.

Matrix A blocks after shifting:

[2 1] [2 1]

[1 2] [1 2]

[1 2] [1 2]

[2 1] [2 1]

Matrix B blocks after shifting:

[2 2] [1 1]

[1 1] [2 2]

[1 1] [2 2]

[2 2] [1 1]

Perform local block multiplication and accumulate the results:

P0: [5 5] + [2 1] \* [2 2] = [9 9]

[4 4] [1 2] [1 1] [9 9]

P1: [4 4] + [2 1] \* [1 1] = [9 9]

[5 5] [1 2] [2 2] [9 9]

P2: [4 4] + [1 2] \* [1 1] = [9 9]

[5 5] [2 1] [2 2] [9 9]

P3: [5 5] + [1 2] \* [2 2] = [9 9]

[4 4] [2 1] [1 1] [9 9]

1. After the iterations, gather the blocks to form the resultant matrix C:

Matrix C:

9 9 9 9

9 9 9 9

9 9 9 9

9 9 9 9

MPI\_Sendrecv\_replace instead of MPI\_Sendrecv:

MPI\_Sendrecv\_replace is used because it allows efficient shifting of the matrix blocks without the need for separate send and receive buffers. The received data can replace the sent data within the same buffer, as each process only needs to store one block of matrix A and one block of matrix B at a time. This simplifies buffer management and saves memory, making it a suitable choice for this application.

* Buffer visualization for P2 and P3 before and after executing MPI\_Sendrecv\_replace:

Before:

Buffer\_P2: [2 1] Buffer\_P3: [2 1]

[1 2] [1 2]

After:

Buffer\_P2: [2 1] Buffer\_P3: [2 1]

[1 2] [1 2]

In this case, both P2 and P3 exchange their matrix A blocks with each other using MPI\_Sendrecv\_replace, ensuring that each process receives the correct block of matrix A for the initial alignment.