Martin Carignan
CSC 548
03/20/2025

Midterm Project

Introduction to SUMMA Dense GEMM:

For my SUMMA implementation I had 3 different stationary matrix versions: stationary A matrix, B matrix, and C matrix.

SUMMA Implementation(Description and Processor Grid are the same for any stationary implementation):

Description:

This algorithm uses three user defined functions: *disbtribute_matrix_blocks(), custom_block(), and summa_stationary_c().*

Processor Grid:

The grid is sized by square rooting the number of MPI processors passed in as an argument. *MPI_Cart_create()* function is used to make a 2D processor grid. The coordinates of the processors are recorded then after using *MPI_Cart_coords()* which will be used later for broadcasting.

The grid has to be conceptually "split" into rows and columns using *MPI_Comm_split().* The first call for this function will split the grid into rows by calculating ***rank / p*** to get the number of rows and calculate ***rank % p*** to get the rank for each processor row in their respective row. The same will be done for *MPI_Cart_split()* used for the column split, except the row numbers are calculated using ***rank % p*** and the ranks for the processors in their respective column is calculated using ***rank / p***.

Stationary C SUMMA:

Distribution:

Local block dimensions are calculated first:

```
int block_m = ceil(m / p);

int block_k = ceil(k / p);

int block_n = ceil(n / p);
```

After that is done, the function *disbtribute_matrix_blocks()*, is used to distribute the local block using the dimensions calculated earlier. The main portion of the function is a loop that calculates the starting location of the local blocks and the index of that location is stored in **displ[]**,

```
displ[i] = coords[0] * local_rows * global_cols + coords[1] * local_cols;
```

which is passed into *MPI_Scatterv()* which is then used later to scatter the different blocks to their intended processor.

However, after calculating the displacement and before scattering the block, a custom data type needs to be made so that we can send the right data from the matrix through *MPI_Scatterv()*. We use *custom_block()* which utilizes the *MPI_Type_vector()* and *MPI_Type_create_resized()* functions to help make a data type that accounts for the stride between chunks of data in the matrix since we are in reality storing it as a 1D array.

<u>Communication</u>:

As mentioned above teh *MPI_Scatterv()* function is used for distributing blocks but the major portion of communication happens in the computation loop:

1. Two temp arrays, the size of blocks for matrix A and matrix B, need to be allocated because these will store the block elements that will be broadcast through their respective communicators.
2. The loop that will iterate through the grid will copy and paste the data from the local matrix to the temp matrix. For matrix A, it will be the blocks who are in the column coordinate number that corresponds to the iteration number, and for matrix B, the row coordinate number that corresponds to the iteration number.
3. After that the A matrices will broadcast their temp matrices to their respective rows and the B matrices will also broadcast their temp matrices to their respective columns in the grid.
4. A *matmul()* function will perform matrix multiplication like normal store the computed values in local **my_C** matrix

```
for(int iter_k = 0; iter_k < p; iter_k++){
    if(coords[1] == iter_k){
      memcpy(temp_A, my_A, my_A_size * sizeof(float));
    }
    if(coords[0] == iter_k){
      memcpy(temp_B, my_B, my_B_size * sizeof(float));
    }
    MPI_Bcast(temp_A, my_A_size, MPI_FLOAT, iter_k, row_comm);
    MPI_Bcast(temp_B, my_B_size, MPI_FLOAT, iter_k, col_comm);
    matmul(temp_A, temp_B, my_C, block_m, block_n, block_k);
  }
```

Result Collection:

After the local C blocks have accumulated their local matrix multiplication, the *MPI_Gatherv()* function is used to store the C blocks into their corresponding location within global C matrix.

Stationary A or B SUMMA:

Distribution:

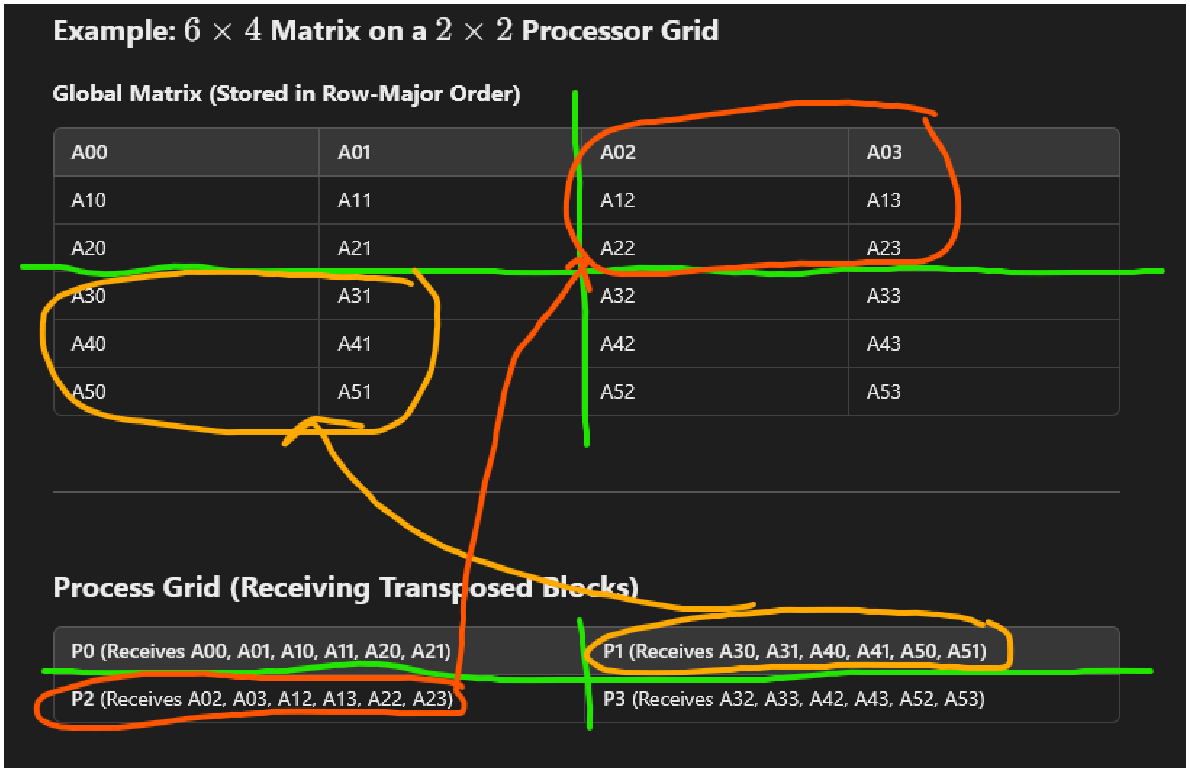
Local block dimensions are calculated first:

```
int block_m = ceil(m / p);

int block_k = ceil(k / p);

int block_n = ceil(n / p);
```

After that is done, the function *disbtribute_matrix_blocks()* is used again, but only for the matrix that will be stationary. The other matrix will be passed through a similar function, *trnsps_distr_mat_block()*, that distributes the transpose of the block assignments. The slight difference in this function is the following:

```
displ[i] = coords[1] * local_rows * global_cols + coords[0] * local_cols;
```

As you can see the coords array elements that represent the column and row are swapped, the rest of the function is exactly the same as *disbtribute_matrix_blocks()*. What this simple change does is it locates the start of the block address in a column-wise manner rather than row-wise like before.



Example: 6×4 Matrix on a 2×2 Processor Grid

Global Matrix (Stored in Row-Major Order)

A00	A01	A02	A03
A10	A11	A12	A13
A20	A21	A22	A23
A30	A31	A32	A33
A40	A41	A42	A43
A50	A51	A52	A53

Process Grid (Receiving Transposed Blocks)

P0 (Receives A00, A01, A10, A11, A20, A21)	P1 (Receives A30, A31, A40, A41, A50, A51)
P2 (Receives A02, A03, A12, A13, A22, A23)	P3 (Receives A32, A33, A42, A43, A52, A53)

<ins>Communication</ins>:

The SUMMA computation loop is similar to the the stationary C implementation, however, in this scenario the matrices being broadcast are C and the non-stationary matrix(choose one).

If stationary A SUMMA is being performed, a temp_B array copies the local B block if a processor's row coordinates match the iteration number and the temp_C array copy and pastes the local C block if a processor's column coordinates match with the iteration number. The temp_B is broadcast along its respective processor columns and the C array is broadcast along its respective processor rows.

If stationary B SUMMA is being performed, a temp_A array copies the local A block if a processor's column coordinates match the iteration number and the temp_C array copy and pastes the local C block if a processor's row coordinates match with the iteration number. The temp_A is broadcast along its respective processor rows and the C array is broadcast along its respective processor columns.

After broadcasting, *matmul()* is called but immediately after, MPI_Reduce() is necessary to accumulate the results calculated from the current iteration. This is because the temp_C array will be what holds the result but upon the next iteration temp_C will replace its values with the local C array.

Result Collection:

Result collection works the exact same way as stationary C implementation.

Tests

Environment:

- Slurm
- Arc cluster
- Milan CPU
- C language
- 4 nodes
- 16 tasks

Cases Performed:

Square Matrices: (times are recorded in the ./outputs folder to view)

```
mpirun -np 16 ./main -m 8192 -n 8192 -k 8192 -s a -v -p

mpirun -np 16 ./main -m 16384 -n 16384 -k 16384 -s a -v -p

mpirun -np 16 ./main -m 8192 -n 8192 -k 8192 -s b -v -p

mpirun -np 16 ./main -m 16384 -n 16384 -k 16384 -s b -v -p

mpirun -np 16 ./main -m 8192 -n 8192 -k 8192 -s c -v -p

mpirun -np 16 ./main -m 16384 -n 16384 -k 16384 -s c -v -p
```

Rectangular Matrices: (times are recorded in the ./outputs folder to view)

```
mpirun -np 16 ./main -m 8192 -n 8192 -k 128 -s a -v -p

mpirun -np 16 ./main -m 16384 -n 16384 -k 128 -s a -v -p

mpirun -np 16 ./main -m 8192 -n 8192 -k 128 -s b -v -p

mpirun -np 16 ./main -m 16384 -n 16384 -k 128 -s b -v -p

mpirun -np 16 ./main -m 8192 -n 8192 -k 128 -s c -v -p

mpirun -np 16 ./main -m 16384 -n 16384 -k 128 -s c -v -p
```

Observations:

<u>Communication Analysis</u>:

(*np = total number of processors, p = number of processors along the length*)

MPI_Scatterv(): messages sent/received = np - 1 sent/received

MPI_Bcast(): messages sent/received = p - 1 per row or column sent/received

MPI_Reduce(): messages sent/received = p per row or column sent/received

MPI_Gatherv(): messages sent/received = np - 1 sent/received

<u>Performance Analysis</u>:

Based on the resulting execution time of my program, it had a noticeably easier time computing tall and skinny matrices compared to square matrices. This is probably due to the tall and skinny matrices having less elements in total that needed to be accumulated to the resulting vector, where the square matrices had much more to compute.

The memory access pattern and load balance is uniform in this implementation. This is because I made it a requirement that the length and width of the matrices have to be a multiple of the total number of processes. I did this for the ease of writing an algorithm that can generalize where to access data and distribute uniform amounts of data.

<u>Conclusion</u>:
After analysis of my implementation of SUMMA Dense GEMM algorithm, what I found was that rectangular matrices were teh quicker ones to compute substantially compared to square matrices and the stationary C was slightly faster that stationary A or B matrices. The reason for the first finding was explained in the Performance Analysis section, but the reason I believe that stationary C SUMMA was faster than the other stationary matrices is because stationary C used less MPI communication than the other two. In stationary A and B, MPI_Reduce() was used at the end of each iteration for the SUMMA computation loop adding O(p) complexity inside the loop, resulting in that for() loops complexity being O(p * p/2). This does add a slight amount of time compared to stationary C's SUMMA for() loop which is O(p). The main challenge of this project was first grasping the SUMMA and GEMM concept so that I could start figuring out how to apply it in code, but then came the challenge in understanding how I would be using MPI cartesians to help me with the implementation. After grasping those concepts it became clearer what I needed to do.