0.1. Methodology

- the matrix A is divided into multiple localA matrices by rows such that each process take a contiguous block of kI/(number of process)
- The whole matrix B (kK * kJ) is broadcasted to all matrices.
- Each process computes its C_local = A_local * B_local using old code from lab1
- The matrix C is gathered at the end

0.2. Analyzing APIs

Blocking	Buffered-Blocking	Non-blocking
slowest since it blocks for every communication operation	higher throughput than Blocking since it can put the data in the buffer and return faster	highest throughput since no communication operation is blocked. However it is hard to implement

Perf: 186.959 GFlops

0.3. Comparing the sequential version with the two parallel versions

Performance	sequential	MPI
1024^3	0.58 GFlops	70 GFlops
2048^3	0.27 GFlops	145 GFlops
4096^3	0.19 GFlops	190 GFlops

The performance greatly differs between different block size because the communication time is neglectable compared to the total time. The larger the poroblem, the more latency is hidden.

0.4. Scalability

np	Performance
1	60
2	108
4	190
8	166
16	111
32	74

the performance increases as np increase but peaked at np=4, and decreases as np get bigger because each process compute is too light relative to the fixed cost of communication.

```
void computeLocalTiledGEMM(float (*a)[kK], float (*c)[kJ], const float (*b)[kJ], int rows)
{
    float tileA[TILE_I][TILE_K], tileB[TILE_K][TILE_J];
    int i, j, k;
    for (int ib = 0; ib < rows; ib += TILE_I)
    {
        for (int jb = 0; jb < kJ; jb += TILE_J)
        {
            float tileC[TILE_I][TILE_J] = {0};

            for (int kb = 0; kb < kK; kb += TILE_K)
            {
                for (i = 0; i < TILE_I; ++i)
                {
                    tileA[i][0] = a[ib + i][kb];
                    tileA[i][1] = a[ib + i][kb + 1];
                    tileA[i][2] = a[ib + i][kb + 2];
                    tileA[i][3] = a[ib + i][kb + 3];
                    tileA[i][4] = a[ib + i][kb + 4];
                    tileA[i][5] = a[ib + i][kb + 5];
                    tileA[i][6] = a[ib + i][kb + 6];
                    tileA[i][7] = a[ib + i][kb + 7];
                }

                for (k = 0; k < TILE_K; ++k)
                {
                    for (j = 0; j < TILE_J; j += 8)
                    {
                        tileB[k][j] = b[kb + k][jb + j];
                        tileB[k][j + 1] = b[kb + k][jb + j + 1];
                        tileB[k][j + 2] = b[kb + k][jb + j + 2];
                        tileB[k][j + 3] = b[kb + k][jb + j + 3];
                        tileB[k][j + 4] = b[kb + k][jb + j + 4];
                        tileB[k][j + 5] = b[kb + k][jb + j + 5];
                        tileB[k][j + 6] = b[kb + k][jb + j + 6];
                        tileB[k][j + 7] = b[kb + k][jb + j + 7];
                    }
                }

                for (i = 0; i < TILE_I; ++i)
                {
                    for (j = 0; j < TILE_J; ++j)
                    {
                        tileC[i][j] += tileA[i][0] * tileB[0][j];
                        tileC[i][j] += tileA[i][1] * tileB[1][j];
                        tileC[i][j] += tileA[i][2] * tileB[2][j];
                        tileC[i][j] += tileA[i][3] * tileB[3][j];
                        tileC[i][j] += tileA[i][4] * tileB[4][j];
                        tileC[i][j] += tileA[i][5] * tileB[5][j];
                        tileC[i][j] += tileA[i][6] * tileB[6][j];
                        tileC[i][j] += tileA[i][7] * tileB[7][j];
                    }
                }
            }

            for (i = 0; i < TILE_I; ++i)
            {
                for (j = 0; j < min(TILE_J, kJ-jb); j++)
                {
                    c[ib + i][jb + j] = tileC[i][j];
                }
```

The function compute matrix multiplication for each thread is the performance bottle-neck. since the values of TILE is optimized for np=4, as the the number of processes increase, the smaller the split array; hence, it increase the overhead for each computation.

0.5. Comparison MPI with OpenMP

Solving this problem with OpenMP is simpler and easier to debug than MPI, as MPI is more prone to out-of-bound errors.
The matrix multiplication section from Lab 1 is reused in Lab 2, along with additional code for splitting matrices.
My performance in Lab 1 was 176, while Lab 2 achieved 190, mainly because MPI runs processes on separate physical cores; potentially reduce contingency
However, this advantage holds primarily for small problem sizes, as MPI introduces significant communication overhead at small problems (OpenMP 171 vs MPI 70)