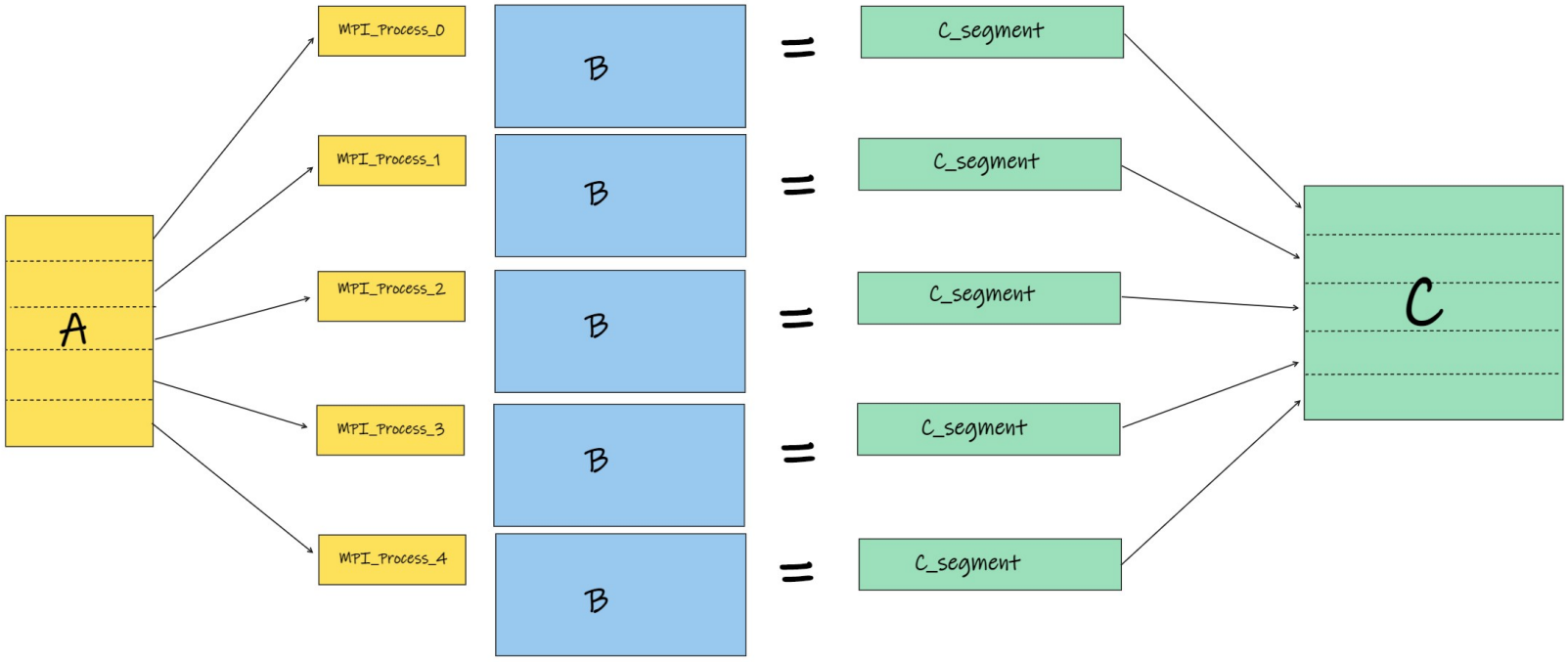
Algorithm

The algorithm multiplies two (not necessarily square) matrices using OpenMP and MPI in C++. The speedup is achieved by partitioning the rows of A for each of the multiplication. First, we partition the *m* rows of A into *k* slices (*k* must divide *m*).

Each slice will contain *m/k* rows of A and each will be on a separate MPI process. Each of these processes will compute the dot product between it's slice of A and the entire matrix B. In this way, if we can allocate one process per row of A, each process will effectively have a matrix-vector product that can be completed in O(n^2) runtime. Since they all occur in parallel, we can do matrix-matrix multiplication in O(n^2) time. This can get difficult to allocate this number of processes when we have a VERY large number of rows.

The diagram below illustrates this well. First we partition A, then each partition performs a matrix multiplication between it's partition and ALL of B. We get the c_segment which is part of the solution. Then we finally combine the c_segments to get the resulting matrix C.



An important thing to note is that each MPI process will be allocated some number of threads. So if we have 5 MPI processes and we input 16 threads, then each of the 5 MPI processes will have 16 threads. Hence, 80 threads total.

Pseudocode:

```
MatrixMultiplication(A,B)
    C = [A.rows, B.cols]
    rank = rank of this process in MPI execution
    numSlices = number of processes in MPI execution
    sliceSize = A.rows/ numSlices
    startIdx = (rank*A.rows)*sliceSize // jump over the previous slices
    sliceArr = &A[startIdx] // sliceArr now points to the address of A associated with this slice

    // perform the standard matrix multiplication between sliceArr and B.
    for i=0 to sliceSize
        for k=0 to A.rows
            r = A[i,k]
            for j=0 to B.cols
                C[i,j] = r * B[k,j]

    // Consolidate C
    // MPI data type with correct offsets to get this ranks contribution
    sliceDataType = C[A.cols*sliceSize]
    localStartIdx // calculate my local start index in C

    if (rank != 0)
        send(C[localStartIdx], sliceDataType)

    else if (rank == 0)
        for sendingRank=1 to numSlices
            receivingIdx =  // calculate the receiving index using rank
            receive(C[receivingIdx], sliceDataType)
```

Space Complexity:

Each MPI process starts out with a complete copy of A and B, but will only have a partial copy of C. However, we still allocate the full size of C for each rank. Also, while we don't use all of A, we still must load it so we can access our rank's slice. Therefore, each rank will have the following complexity:

A - mxn

B - nxp

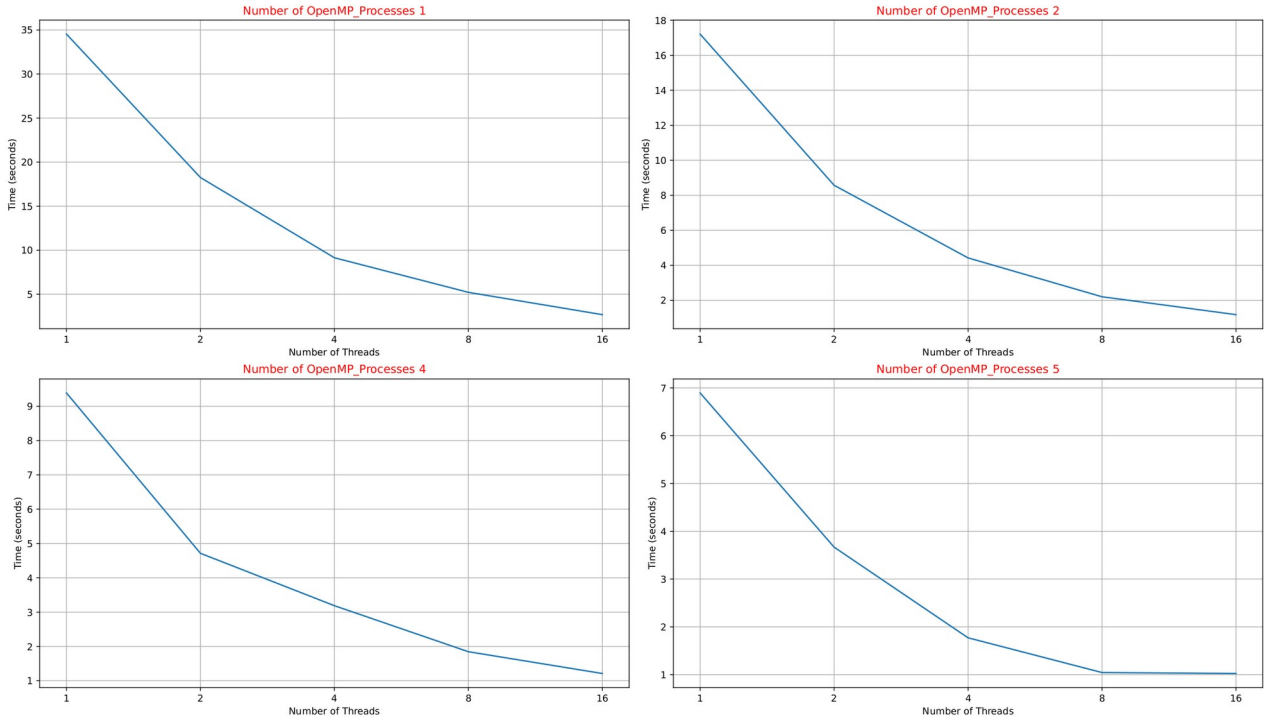
C - mxp

Therefore, our space complexity is O(m*n + n*p + m*p). To simplify, we can let *n* be the largest dimension and say that the space complexity is O(3*n^2) = O(n^2).

Runtime:

I achieved the following runtimes on matrices of size A=(1000,10000) and B=(10000,1000).

MPI_Partitions	OpenMP_Threads (per MPI partition)	Total processes	Time (seconds)
1	1	1	34.53
2	1	2	18.25
4	1	4	9.14
8	1	8	5.22
16	1	16	2.68
1	2	2	17.21
2	2	4	8.57
4	2	8	4.42
8	2	16	2.2
16	2	32	1.18
1	4	4	9.39
2	4	8	4.71
4	4	16	3.19
8	4	32	1.85
16	4	48	1.21
1	5	5	6.89
2	5	10	3.67
4	5	20	1.77
8	5	40	1.04
16	5	80	1.02



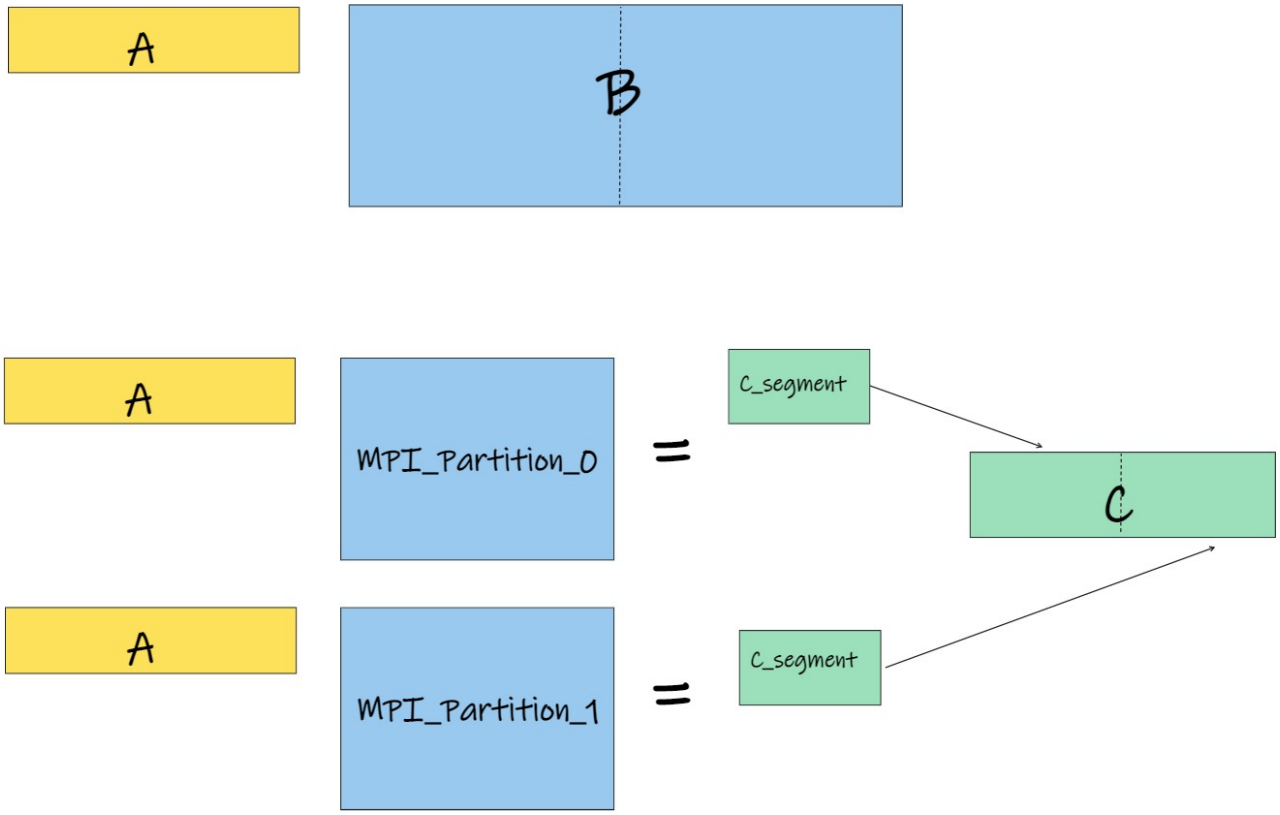
IMPORTANT NOTE: The graphs above are logarithmically increasing along the x-axis. That is to say that the number of OpenMP threads are doubling while the time increases linearly. Therefore, if we were to scale it one to one, the time would drop off much more quickly.

Shortcomings:

My algorithm asssumed the matrices are relatively square. In particular, the rows of A should be at least as large as the cols of A. This is because we are only partitioning in terms of rows of A. Suppose the extreme case where we are multiplying A (1x10000) and B (10000x10000), then the algorithm will be no faster than the serial version. Because it will all run on a single slice on one MPI process.

A possible solution to this issue would be to check the number of cols of B and the number of rows of A and see which is larger. If the rows of A is larger, then partition as we mentioned above. If the cols of B is larger, then partition using the cols of B instead. In other words, we take each column of B and dot product it with the matrix A.

The diagram below shows how this can be done. In this case, it's clear that the rows of A is much smaller than the columns of B. Hence, we can split the columns of B into 2 MPI processes. This would give us a much better result.



This likely still wouldn't be as efficient as the tile method, but it does mitigate the biggest issue in my algorithm.