Exercise 4: Boolean Matrix Multiplication

1. *Theory and applications of BMM:*

In general Boolean matrices are called "Logical" and can be used in several applications in majors like mathematics and engineering. Some of them are "*Systems of linear equations*", "*Computer graphics*", "*Games*" etc.

About graphics, the logical conjunction and disjunction can be used to find features in color images, by thresholding the colors Red, Green and Blue separately.

Another example that BMM can be used for is the context – free grammars (CFG). The first CFGs parses had execution times of $O(gn^3)$ where g is the size of CFG and n the length of the input string. A faster CFG parsing can be accomplish by a fast BMM.

In addition, BMM could also be used in our 1st Exercise, "Triangle Counting".

2. *Codes – Implementations:*

For generating arrays, I used a Matlab script that can be found on GitHub.

To make sure that I was having the correct results, I run some tests with small sizes and check my results with those of Matlab. An example will be on Github.

After creating the sequential version of BMM and blocking BMM, I implement the filter in those codes. The filter was being calculated in the Matlab script as well. By implementing this filter there was a massive difference in execution times, as you can see on ***Figure 1***.

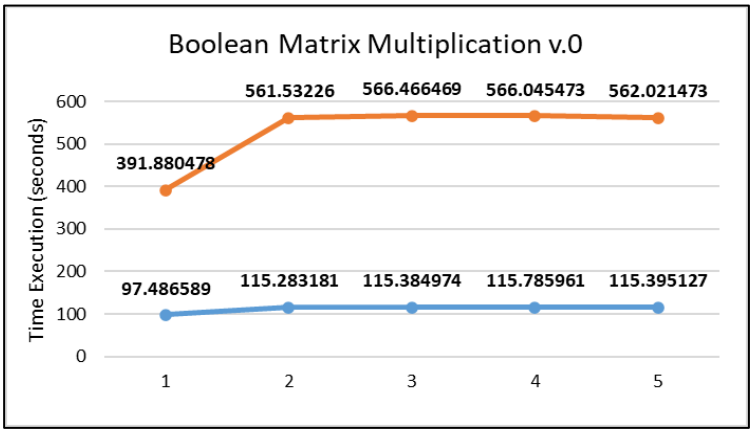
For the blocking, I created two functions *(blockSparse, convertBS)*. The first one returns all the data that I need to do the blocking and the second function converts the sparse matrix to block sparse matrix. The factor that I consider for choosing was the size of the array. The reason is explained below where the comments of execution times are.
The block sparse multiplication is similar to the sparse multiplication.
To begin with, the sparse multiplication iterates through CSR and finds the positions ([i , j]) of the product "C" that would be equal to 1.
The only difference of the block sparse multiplication is that it begins to iterate through BCSR (blocks), instead of CSR and then uses the CSR to find the inner elements of each block. Thus, the product "C" is being calculated block after block, instead of one element at each time. This helps the performance when there are a lot of blocks without non zeros.

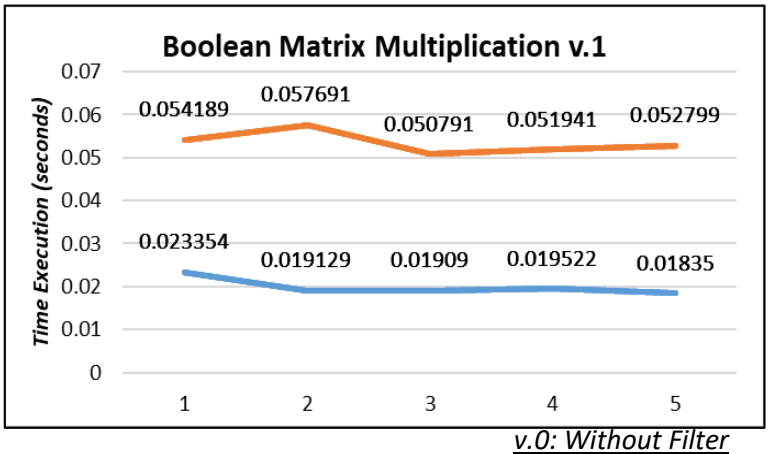
For parallelizing I used MPI and OpenMP for each version. The part that is being parallelized is where the multiplications are being calculated. MPI and OpenMP help to decrease even more the execution times on some version. The idea on combining OMP and MPI was to use MPI processors for the blocks and OMP for the inner elements of each block.

3. Execution Times:

I used Google Colab to run my codes, because docker had an update and won't start after it. "Ipynb" file will be on GitHub.

- *Non filter and filter - Non Block Versions **(Figure 1)**:*





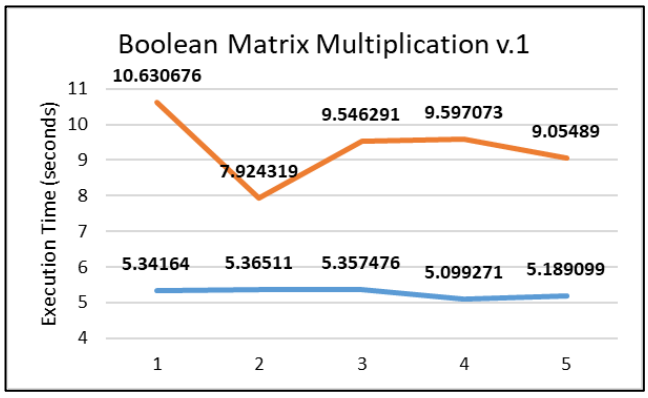
v.0: Without Filter
v.1: With Filter

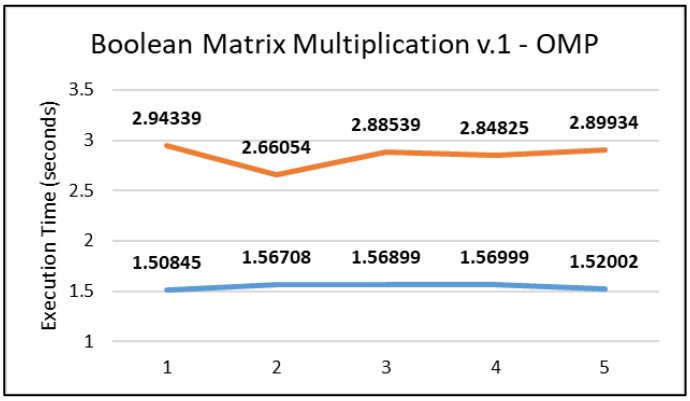
50000 Rows
100000 Rows

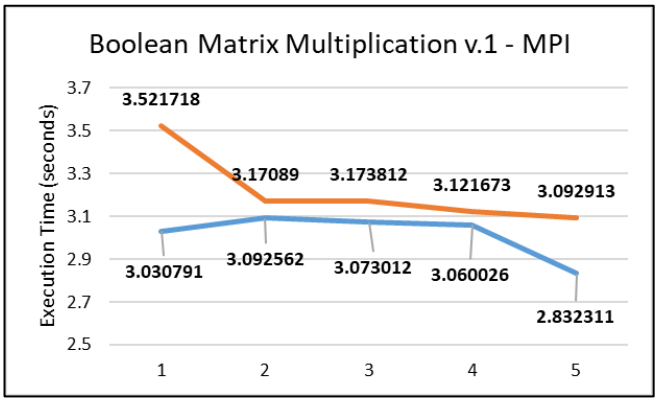
From the graphs it's clearly that the version with the filter executes much faster, as we mention earlier. By adding the filter, the function calculates only those elements where the filter is equal to 1.

- *Sequential and parallel – Non Block Versions **(Figure 2)**:*

From these graphs we can see the executions times from the sequential version with the filter and the parallel ones. By adding more processors and threads, we get the results that we expected. Also, from these tests we make sure that our sequential version has an execution time close to those that Matlab script has too.
In addition, by adding more threads or processors, the execution time decreases even more.



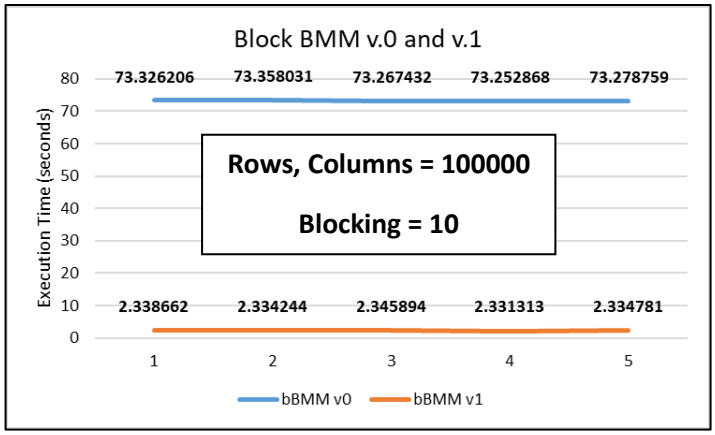




MPI: 2 Processors, OMP: 2 threads
MPI: 3 Processors, OMP: 3 threads

3000000 Rows
6000000 Rows

- *Non filter and filter – Block Versions **(Figure 3)**:*



At this graph we see the execution times of the block BMM versions without and with filter. The version with the filter, it's definitely faster in comparison with the version without the filter.
But, if we compare it with the ***Figure 1***, we can see that the blocking contribute to decrease the times only for the version without the filter. The reason for that is because I wasn't able to apply the filter in the multiplication of inner elements.

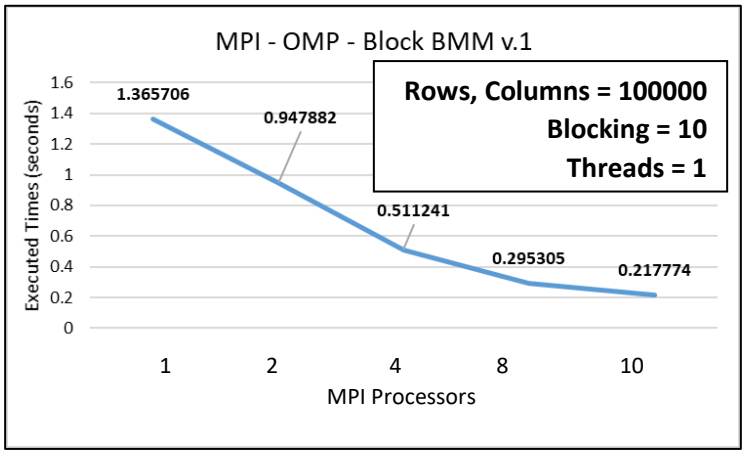
```
for (int i=0; i<f_bcsr.block_rows; i++)
    for (int j=fb csr row[i]; j<fb csr row[i+1]; j++)
```

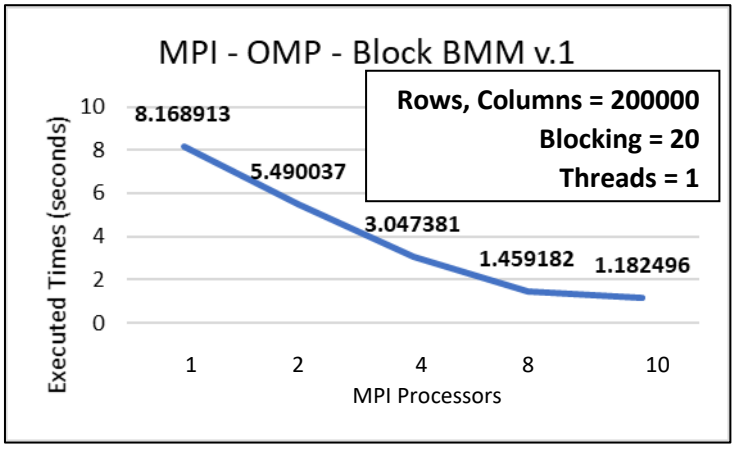
Block iteration with filter.

```
for (int k=blocking*i; k<(blocking*i)+blocking; k++)
    for (int h=blocking*j; h<(blocking*j)+blocking; h++)
        if (multiply(k, h, csr row, csr col, csc row, csc col)){
```

Inner Block multiplication without filter.

- *Two levels of Parallel – Block Versions **(Figure 4)**:*





Because of the problem that I explained before, I wasn't able to run tests with big arrays. Although I made some runs with smaller sizes to see the differences between sequential and two levels of parallel. Also, about *blocking size*, I choose it based on the idea of having many blocks with small size, so that the inner multiplication, which is not filtered, won't affect the performance too much.

If we compare the left graph with the one in ***Figure 3***, we can see that even with 1 processor it has a bit of speed up. This is because of the OMP and its optimization flag. Moreover, by increasing the number of processors, execution times are being decreasing as it was expected. The right graph is the same as the left, just for doubled size. In both of them, the speed up is approximately the same.

GitHub Link: *https://github.com/mariostavr/Boolean-Matrix-Multiplication*
GitHub: Codes, Sources, Tests, PDF