Parallel & Distributed Boolean Matrix Multiplication Using Blocks

Petridis Konstantinos 9403 | Gitopoulos Giorgos 9344

Parallel & Distributed Systems @ ECE AUTh, September 2021

Abstract

Boolean Matrix Multiplication (BMM) has some significant applications in engineering problems. First of all, it is utilized in constraint networks which are represented by a graph called a primal constraint graph. In such graphs, each node represents a variable and the arcs connect all nodes whose variables are included in a constraint scope of the problem. The absence of an arc between two nodes indicates that there is no direct constraint. Constraint deduction-inference from initial set of constraints is accomplished through **BMM**. It is also used in context-free grammar (CFG), whose rules dictate that a single nonterminal symbol may produce a string of terminals and/or nonterminals. **BMM** is specifically used in context free parsing via union calculation formulas and convolution of boolean vectors. Other applications include coordinate transformation, solution of linear system equations, transitive closure and network theory.

1. Introduction

This project exhibits the implementation of a fast, high-performance **BMM** algorithm for **sparse** matrices, as well as the tools deployed to make it more efficient. **Blocking** is introduced as a practical way of accelerating **BMM** and facilitating its parallelism, while **MPI** and **OpenMP** are used to distribute and parallelize the computations respectively. The code of the project was written in **C++** and can be found **here**.

2. BMM

2.1. Classic Algorithm

We assume two matrices $A = A(I, K) = [a(i, k)]$ with row index set I and column index set K and $B = B(K, J) = [b(k, j)]$ with index sets K and J. Let $C = C(I, J) = [c(i, j)]$ be the boolean product

$$C = AB$$

where

$$c(i, j) = \vee_{k \in K} a(i, k) \wedge b(k, j)$$

with $\vee$ and $\wedge$ being the logical disjunction (or) and conjuction (and) symbols respectively. Element $c(i, j) = 1$ if and only if there exists k in the search domain K such that $a(i, k)b(k, j) = 1$, i.e., there is a match between the binary vector in row i of A and the binary vector in column j of B.
In our project we perform the *BMM* between two matrices A and B in *CSR* (*Compressed Sparse Row*) and *CSC* (*Compressed Sparse Column*) format respectively, according to the following algorithms:

Algorithm 1 Boolean Matrix Multiplication

```
1: procedure BMM(A, B, C)
2:     for rowA = 0 : numOfRows(A) do
3:         for colB = 0 : numOfCols(B) do
4:             C(rowA, colB) ← mult(rowA, colB, A, B)
```

Algorithm 2 Boolean Row-Column Multiplication

```
1: procedure MULT(rowA, colB, A, B)
2:     if search(k: A(rowA, k) = B(k, colB) = 1) then
3:         return true
4:     else
5:         return false
```

For every row of A we choose the columns of B that corresponds to each column of A and we look for a pair such that $A(rowA, k) = 1$ and $B(k, colB) = 1$. In regular matrix multiplication we would continue counting the number of these pairs, but in *BMM* the external $+$ operation is replaced by the logical $\vee$, so as soon as we locate one corresponding pair with *logical conjunction* equal to one, we stop and move on to the next row of A.

2.2. Masked Algorithm

In the generalized form of *BMM*, a filter matrix F is used, and the operation transforms to

$$C = F \odot (AB)$$

so that $C(i, j)$ can be true only if $F(i, j) = 1$. The corresponding algorithm is presented below:

Algorithm 3 Masked Boolean Matrix Multiplication

```
1: procedure MASKEDBMM(F, A, B, C)
2:     for rowF = 0 : numOfRows(F) do
3:         for colF in non-zero indices of rowF do
4:             C(rowF, colF) ← mult(rowF, colF, A, B)
```

Note that the non-zero indices of a specific row of a *CSR* matrix can be accessed immediately using the *row pointer* of *CSR* form.
The multiplication of *Algorithm 3* is called *masked-BMM*. Obviously, *masked-BMM* has lower computational complexity, especially in the case that F is sparse. Although both *BMM* and *masked-BMM* routines are implemented in the project, the second was preferred for convenience in testing.

3. Blocking

Based on the ideas of *Buluç et al.* [1], who stored the matrix in blocks (*CSB - Compressed Sparse Blocks*) in order to perform efficient sparse matrix-vector multiplication, we tried to apply a block storing format to be used in the *BMM*. We ended up in the **BCSR (Block Compressed Sparse Row)** format almost as it is described by *Vuduc et al.* [2].

We decided to use a *high-level block row pointer* array and a *high-level block col index* array in order to store the non-empty blocks. These two arrays store the non-empty blocks just like *CSR* stores the non-zero elements of a matrix. *High-level block row pointer* specifies the number of non-empty blocks in each block-row and *high-level block col index* stores their block-column indices. In a $m \times n$ matrix with block size b, each block-row consists of m/b rows and each block-col consists of n/b cols. We also use a *non-zero block index* array that directly gives the position of the non-empty blocks and a *block non-zero counter* array that stores the accumulative sum of the inner non-zero elements of the blocks.

According to **space complexity**, if m is the number of matrix rows, b is the block size, nnz is the number of non-zero elements and $nnzb$ the number of non-empty blocks, then *high-level block row pointer* and *high-level block col index* allocate $((m/b) + 1) \times sizeof(int)$ and $nnzb \times sizeof(int)$ bytes respectively. *Non-zero block index* and *block non-zero counter* both allocate $numBlocks \times sizeof(int)$ bytes.

In the following example, there is a 9×9 array with $b = 3$. The high-level *BCSR* format is:

$$HL\_blockRowPtr = [0, 2, 3, 5]$$

$$HL\_blockColInd = [0, 2, 1, 0, 2]$$

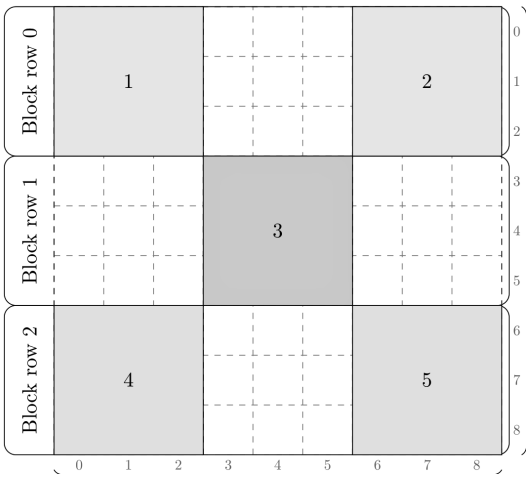

Figure 1: *Blocked matrix example*

In the inner-block level, the non-zero elements are stored in *CSR* format in *low-level block row pointer* and *low-level block col index* successively block by block. The *low-level block row pointer* allocates $nnzb \times (b+1) \times sizeof(int)$ and the *low-level block col index* allocates $nnz \times sizeof(int)$ bytes. We also implemented the **BCSC (Block Compressed Sparse Column)** format that stores the matrix in both levels in *CSC* format, just like *BCSR* stores it in *CSR* in both levels. Both *BCSR* and *BCSC* will be used in the *BMM* using blocks.

4. BMM using Blocks

In this section, our algorithm of *BMM* using *BCSR* and *BCSC* matrices will be explained. *BMM* using blocks has been implemented for both unmasked and masked multiplication, but as we have already mentioned, we will focus on masked *BMM* for the purposes of this report.

The main idea of *BMM* using blocks is to avoid unnecessary computations of empty block-rows and block-cols. We present the following algorithms, where $M_{i,j}$ is the block of block-row i and block-col j of a matrix M:

Algorithm 4 Masked BMM using Blocks

```
1: procedure MASKEDBLOCKBMM(F, A, B, C)
2:     for i = 0 : numOfBlockRows(F) do
3:         for j in non-empty indices of block-row i do
4:             C_{i,j} ← blockMult(i, j, F, A, B)
5:             C_{i,j} ← addDisplacements(C_{i,j}, i, j)
```

Algorithm 5 Masked blockRow - blockCol Multiplication

```
1: procedure BLOCKMULT(i, j, F, A, B)
2:     C_{i,j} ← nullBlock
3:     for search(all k: A_{i,k} ∧ B_{k,j}) do
4:         C_{i,j} ← maskedBBM(F_{i,j}, A_{i,k}, B_{k,j}, C_{i,j})
5:     return C_{i,j}
```

Algorithm 6 Masked Boolean Block Multiplication

```
1: procedure MASKEDBBM(F_{i,j}, A_{i,k}, B_{k,j}, C_{i,j})
2:     maskedBMM(F_{i,j}, A_{i,k}, B_{k,j}, C_{i,j})
3:     return C_{i,j}
```

Matrices F and A have been blocked in $BCSR$ format, while B in $BCSC$. According to the previous algorithms, two separate multiplications take place. A high-level multiplication uses the high level arrays that described in the previous section in order to locate the possible non-empty blocks $C_{i,j}$ and the low-level multiplication of the corresponding blocks $A_{i,k}$ and $B_{k,j}$ follows just like the original *BMM*. The resulting blocks $C_{i,j}$ are added to the sparse matrix C in COO format after having their indices fixed by adding the suitable displacements. The main advantage of this way of computation, is that it directly locates possible non-empty blocks using $BCSR$ and $BCSC$ formats.

5. Parallel Implementation

We tried to accelerate our code parallelizing it using **OpenMP**. Each iteration of the outer loop of *Algorithm 4* computes a block-row of the product matrix C. Each iteration of the inner loop computes a specific block of the relevant block-row of C. We decided to parallelize the inner loop, so that the computation of a specific block-row of C will be executed in parallel, while the block-rows will iterate sequentially.

However, the inner-loop of the sequential algorithm adds the product block directly to the result matrix. This can lead to **data-races** in the parallel version, as more than one threads may modify the data structure of matrix C at the same time. To avoid this problem, in every iteration of the outer loop we create a *vector* to store the specific block-row of C and each thread writes

the resulting block in a different index of the *vector*. When the inner loop is completed, we sequentially transfer the resulting block-row *vector* to the data structure of matrix C.

6. Distributed Implementation

Another attempt for improving, is the distributed implementation using *MPI*. After declaring the total number of working processes, $process\ 0$ equally distributes the block-rows of F and A to all the processes, while B is broadcasted. Each process, after blocking its local matrices, performs the operation

$$C_i = F_i \odot (A_i B)$$

where C_i is the local result of the process i and F_i, A_i the chunks of block-rows of the original matrices that were distributed to process i. The results C_i are gathered back to $process\ 0$ and stored to the product matrix C. After distribution, the chunk offsets are removed in order to perform the *BMM* and after gathering in $process\ 0$ they are restored.

7. Parallel & Distributed Implementation

The final version of the project attempts to merge *parallel* and *distributed* implementations to gain the maximum speedup. The matrices are distributed exactly as described in the previous section, and the local multiplication takes place in each process according to our *parallel algorithm*. In this way we can provide two levels of parallelism. We initially distribute the computation of matrix C in different processes that work in parallel and each process parallelizes its own computations using its available threads.

8. Benchmarking

We tested all four versions of *BMM* using blocks (sequential, parallel, distributed, parallel & distributed) on *AUTh HPC* (*Intel Xeon E5-2630 v4*).

8.1. Datasets

A *MATLAB* script was implemented to randomly generate square sparse matrices with given density. Three matrices (*F*, *A* and *B*) are generated and stored in files in *COO* form. Also, *BMM* product matrix *C* is computed by the script and is stored in a file too. The input matrix files are used for the testing of our *C++* code, while a tester was implemented to confirm the validation of the result by comparing it with the stored matrix *C* that was computed in *MATLAB*.
In order to test the performance of our code, we used the randomly generated matrices with the following properties:

Table 1: *Properties of the input matrices*

	m	*n*	*nnz*
F	2,000,000	2,000,000	9,999,990
A	2,000,000	2,000,000	9,999,984
B	2,000,000	2,000,000	9,999,991

After testing with matrices of various sizes, it was noticed that *m* and *n* does not affect the performance significantly, while the number of non-zero elements (*nnz*) is the property that mainly defines the computational load. However, the purpose of this project is to present the improvement of the parallel and distributed versions in comparison with the sequential. This improvement is similar for every matrix properties, so we choose to analyze the results of the specific dataset only.

8.2. Block Size

Block size *b* was set to the value 250,000 (optimal value, found after testing) for all three matrices, so that each block-row consists of 8 blocks and the total number of blocks per matrix is 64. We have to mention here, that according to the structure of our parallel version the fact that each block-row consists of 8 blocks, adds a restrain in the improvement using more than 8 threads, as only 8 blocks can be computed in parallel. For a lower value of *b* the possible improvement of parallelism might be greater, but the overall performance of the sequential algorithm gets worse.

8.3. Pre-Processing

In every version of the code, operations related to reading of input data and matrix format transformation (*COO* to *CSR* or *CSC*, CSR to *BCSR*, *CSC* to *BCSC*, etc) are considered as preprocessing and their execution time is not included in the *BMM* time. Moreover, operations of pre-processing like blocking, offer margin of parallelism, but it was out of scope of our project, as we just focus on *BMM* performance. We just note that the average blocking time of our input matrices was 0.31 seconds and the average reading time was 3.9 seconds per matrix.

8.4. Results

8.4.1. Sequential Version

The **sequential** version of *BMM* using blocks was executed in 4.1212 seconds.

8.4.2. Parallel Version

The execution times of the **parallel** version with respect to the used number of threads are presented in the following table:

Table 2: *Execution times of parallel version*

number of threads	*2*	*4*	*8*	12	16
execution time (sec)	1.47065	0.939296	0.525944	0.619713	0.623482

8.4.3. Distributed Version

The execution times of **distributed** version (including communication times) with respect to the selected number of separate nodes are presented below:

Table 3: *Execution times of distributed version*

number of nodes	*2*	*4*
execution time (sec)	3.13639	2.63577

8.4.4. Parallel & Distributed Version

The corresponding times of the **parallel & distributed** version are included in the following tables:

Table 4: *Execution times of hybrid version using 2 nodes*

number of threads per node	*2*	*4*	*8*	12	16
execution time (sec)	2.38936	2.01211	1.87237	1.88196	1.93625

Table 5: *Execution times of hybrid version using 4 nodes*

number of threads per node	*2*	*4*	*8*	12	16
execution time (sec)	2.74707	2.57728	2.21365	2.10181	2.02895

8.4.5. Speedup

The speedup of all three versions according to the sequential version are presented in the following diagrams:

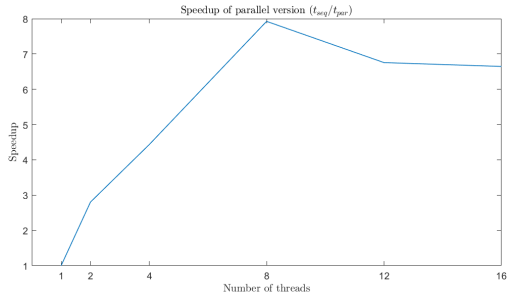

Figure 2: *Speedup of parallel version*

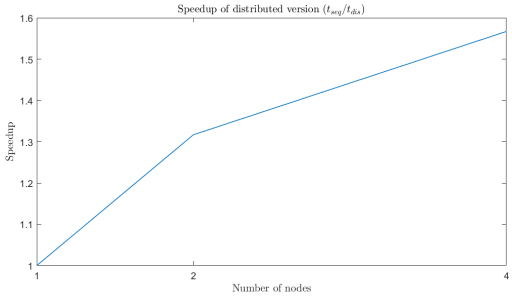

Figure 3: *Speedup of distributed version*

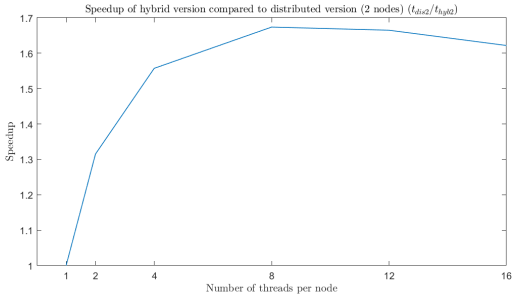

Figure 4: *Speedup of hybrid version compared to distributed (2 nodes)*

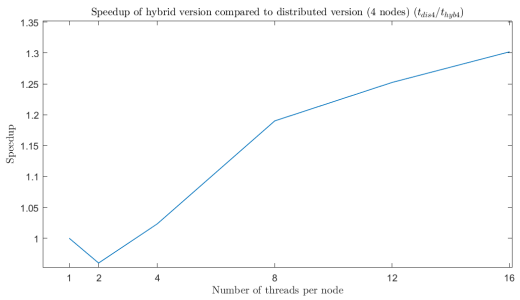

Figure 5: *Speedup of hybrid version compared to distributed (4 nodes)*

9. Discussion

First of all, the **parallel** version offers great improvement in comparison to the sequential computation of *BMM* using blocks, as we can clearly see in *Figure 2*. The results obtained confirm our previously mentioned theoretical approach (8.2) on the speedup limit as we clearly see that the saturation occurs after the number of 8 threads.
The **distributed** version presents visible improvement too, according to *Figure 3*, but it was not able to be tested with more than 4 nodes due to lack of resources.
The **hybrid** version, according to *Table 2*, *Table 4* and *Table 5* did not achieve to improve the parallel version efficiency, but it clearly improves the distributed version (*Figure 4* and *Figure 5*). We can notice that the usage of threads in order to parallelize the computations of each node separately, accelerates even more the distributed version of our code.

10. References

[1] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, Charles E. Leiserson. Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks. In SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architecturesAugust 2009, Pages 233–244.

[2] R. Vuduc, J.W. Demmel, K.A. Yelick, S. Kamil, R. Nishtala, B. Lee. Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, Page 26.