Parallel and Distributed Systems
Sparse Boolean Matrix Multiplication

Θεόδωρος Κατζάλης
AEM:9282
katzalis@auth.gr

13/1/2021

1 Introduction

Boolean Matrix Multiplication is a well known challenge in the computer science scope. The applications span to triangle counting, graph theory, digital signal processing and even cryptography including factoring integers and the algebraic attacks on stream ciphers [1]. The one application that we are most familiar with is the first one, that we have encountered with the first assignment of this course. Briefly, we will attempt to explain it. So, let A be an adjacency matrix of a directed graph G(V,E). Then this matrix, that is represented as a boolean one, can convey the information if the nodes `ith` and `jth` are connected via the check `A[i][j] == 1`. Thus the entry `ij` of A^k counts the number of walks from `i` to `j`. To check if there are triangles or not within the graph, we can use boolean matrix multiplication to compute A^3 and check if the diagonal has a nonzero entry [2].

2 Version analysis

For the Boolean Matrix Multiplication we used four versions: the serial, the OpenMP, the OpenMPI and the Hybrid (OpenMPI/OpenMP). For all of these versions, blocking of the matrices is designed to make use of the inner block matrix multiplication with lookup tables. This optimization isn't though applied but the blocking structure remained causing eventually some bottlenecks that will be discussed further below.

Boolean Matrix Multiplication (BMM), in its naive version, it is a very simple operation, that can be calculated using only for 3 loops. The problem is when the size of the matrices increase. Then for sparse arrays, compressed data structures are utilized. In our case, we are interested for masked BMM, thus instead of computing $C = A \times B$, we have $C = F \odot (A \times B)$, where F is the mask. The mask, with simple terms, indicates which values of the resultant product matrix we are interested beforehand, to avoid computing the whole product. The biggest advantage of BMM opposed to the common matrix multiplication is the redundancy of computing all the inner products of the row-column multiplication. For example to calculate `C(i,j)` with `F(i,j) = 1`, commonly speaking we need the `sum(A(i,k) * B(k,j) for k in 0..n)`. For BMM, this translates to $\wedge_{k\in 0..n} A(i,k) \vee B(k,j)$. Thus the goal is to find the first common element between two lists and then stop iterating over the rest elements. If found, then the result will be `True`. If there aren't common elements then the result for the `(i,j)` will be `False`.

For our sparsed blocked BMM implementation, we decided to use the following scheme: A -> CSR, B-> CSC and F->CSR. The CSC format of the B matrix instead of having all three the same one, is a cache-friendly choice. Since, we are dealing with boolean matrices, we need only two arrays for the compressed formats to spot the `True` (integer 1) values. The pointer array of the CSR structure keeps track the non zero elements row wise and the CSC structure keeps track the non zero elements column wise. Eventually the BMM can be computed by checking the first common element between two lists. So, we need the columns indices of the `ith` row and the row indices of the `jth` column. These two lists can be provided in a contiguous memory layout using CSR for the A and CSC for the B. Another very useful feature is that these two lists are sorted. So the problem at the end of the day is to find if there is at least one common element in 2 sorted lists. A two pointer arithmetic technique was used for that purpose.

As we stated previously, a block structure is designed for the matrix multiplication. That implies that we decompose the matrices to a number of blocks, creating a grid. Now the BMM can be calculated iterating over the blocks as a new higher abstraction layer. To accomplish that a new data structure should be used to locate the blocks of the grid. Instead of 2 arrays for the CSR/CSC (CSX) scheme, we are going to use 4 arrays for the BSR/BSC (BSX) scheme.

2.1 BSX - Blocking CSX

To comprehend the new blocking data structure, we can consider the following example. We assume that we created a grid with number of blocks 2x2. The `BLOCK_ROW_INDEX` is responsible to keep track the non zero blocks per block row (here we have 2 total block rows) and the `BLOCK_COL_INDEX` is responsible for the block column indices (available values: 0,1). Then the `ROW_INDEX` scans the array block, row wise, to find the non zero elements in the inner blocks. Its size is equal to `BLOCK_SIZE_Y * NON_ZERO_BLOCKS + 1`. So the 0-2 indices' range of the `ROW_INDEX` is the CSR of the first non zero block, the 2-4 maps to the second one, and the 4-6 maps to the last. The values of `ROW_INDEX` indicate the indices of the `COL_INDEX` to find the columns within the block. It should be noted that the `COL_INDEX` is adjusted to show the indices in the block level, while the `ROW_INDEX` tracks all the non zero values.

This example described the BSR (the blocked CSR). The BSC (blocked CSC) can be deduced similarly.

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

$$\begin{aligned}
\text{BLOCK\_ROW\_POINTER} &= \begin{pmatrix} 0 & 1 & 3 \end{pmatrix} \\
\text{BLOCK\_COL\_INDEX} &= \begin{pmatrix} 1 & 0 & 1 \end{pmatrix} \\
\text{ROW\_POINTER} &= \begin{pmatrix} 0 & 0 & 1 & 2 & 3 & 3 & 4 \end{pmatrix} \\
\text{COL\_INDEX} &= \begin{pmatrix} 0 & 0 & 1 & 0 \end{pmatrix}
\end{aligned}$$

2.2 Algorithms

These are the algorithms for our serial version that we strive to optimize with our parallel versions. Regarding the last ones, the work of each task is assigned to a subset of rows of the final resultant matrix.

Algorithm 1 Outer block BMM

```
1:  BSX bcsrA ← csr2bcsr(csrA)
2:  BSX bcscB ← csc2bcsc(cscB)
3:  BSX bcsrF ← csr2bcsr(csrF)
4:  CSX csrC ← 0
5:  for blockY ← 1, numBlocksRows do
6:      for idBlockX ← bcsrF.blockPointer[blockY], bcsrF.blockPointer[blockY+1] do
7:          blockX ← bcsrF.idBlock[idBlockX]
8:          blockYRow ← bcsrA.getblockRow(blockY)
9:          blockXCol ← bcscB.getblockCol(blockX)
10:         csrFblock ← bcsrF.getblock(blockY,blockX)
11:         csrCblock ← bmmBlockRowCol(blockYRow,blockXCol, csrFblock)
12:         csrC ← appendCblock(csrCblock)
13:     end for
14: end for
```

Algorithm 2 bmmBlockRowCol() (Inner block BMM)

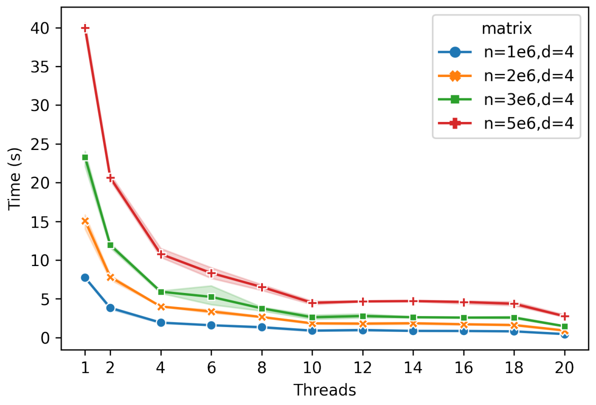
```
1: $commonBlocks \leftarrow$ intersection(blockYRow,blockXRow)    ▷ Find common id blocks of block lists
2: $CSX\ csrCblock \leftarrow 0$
3: for $i \leftarrow$ commonBlocks do
4:     $csrA \leftarrow$ getBlock(i)
5:     $cscB \leftarrow$ getBlock(i)
6:     $csrC \leftarrow$ bmm(csrA,cscB,csrF)
7:     $csrF \leftarrow$ updateMask(csrC)
8:     $csrCblock \leftarrow$ updateC(csrC)
9: end for
```

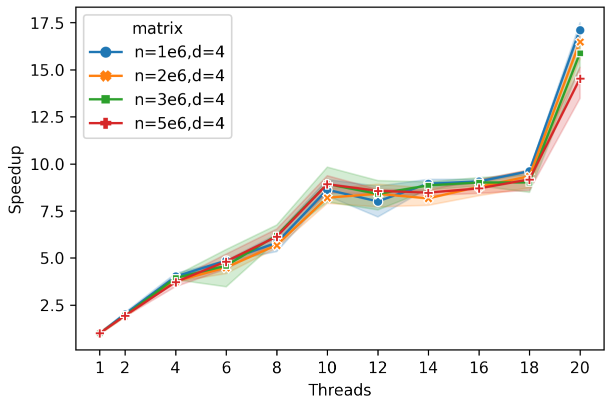
Algorithm 3 bmm() (CSR-CSC BMM)

```
1: $CSX\ csrA \leftarrow$ mm2csr(A)    ▷ Convert Matrix-Market to CSR
2: $CSX\ cscB \leftarrow$ mm2csc(B)
3: $CSX\ csrF \leftarrow$ mm2csr(F)
4: $CSX\ csrC \leftarrow 0$
5: $nnz \leftarrow 0$
6: for $i \leftarrow 1$, rows do
7:     for $j \leftarrow$ csrF.pointer[i], csrF.pointer[i+1] do
8:         $row \leftarrow$ csrA.getRow(i)
9:         $col \leftarrow$ cscB.getCol(j)
10:        if hasCommon(row,col) then    ▷ Two pointer technique
11:            $csrC.col \leftarrow$ csrF.col[j]
12:            $nnz \leftarrow$ nnz+1
13:        end if
14:    end for
15:    $csrC.row \leftarrow$ nnz
16: end for
```

3 Performance data

Regarding benchmarking, four different square sparse matrices have been used with number of rows/columns (n) 1,2,3 and 5 millions with the fixed number 4 of non zeros values (d) per row. The number of blocks were defined with respect to the maximum number of tasks that will be allocated in the row dimension. Thus, for the OpenMP and OpenMPI versions, squared blocks were created with dimensions 20x20 and for the hybrid version, non-square blocks were created with dimensions of 64x1 (block dimensions = the number of blocks != block size) [1] . Non-squared blocks are used for performance issues as discussed further below. It should be noted that the experiments are executed three times, thus error bars are included.

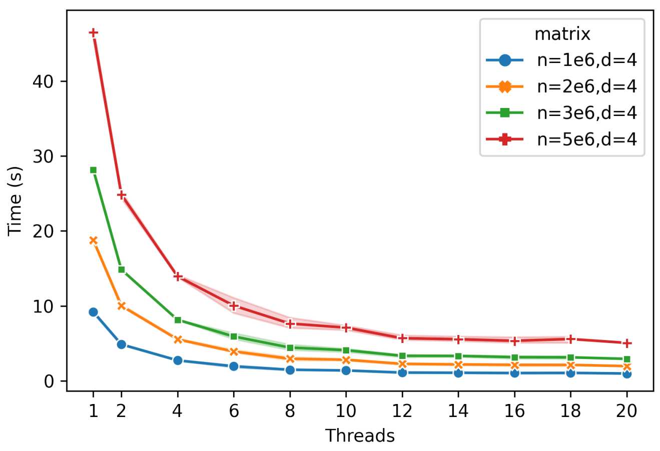


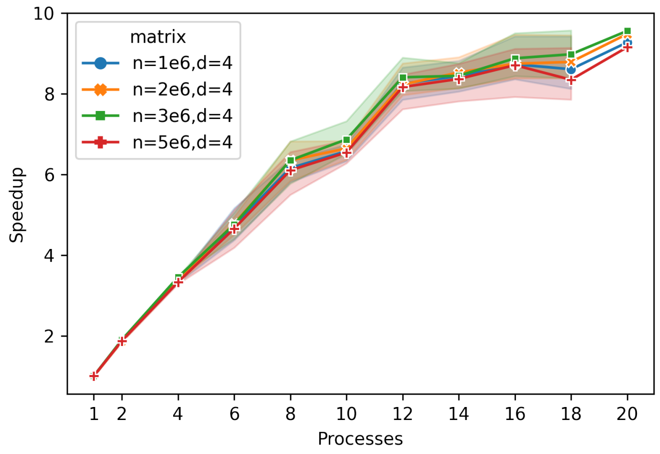


Threads		1	2	4	6	8	10	12	14	16	18	20
n=1e6	Time	7.74	3.83	1.93	1.59	1.33	0.89	0.97	0.86	0.85	0.80	**0.45**
	Speedup	1	2.00	4.00	4.85	5.78	8.60	7.95	8.94	9.06	9.61	**17.10**
n=2e6	Time	15.09	7.82	4.01	3.36	2.65	1.84	1.80	1.85	1.72	1.61	**0.91**
	Speedup	1	1.92	3.76	4.48	5.67	8.18	8.37	8.14	8.76	9.13	**16.45**
n=3e6	Time	23.27	11.94	5.92	5.25	3.77	2.62	2.78	2.62	2.58	2.58	**1.46**
	Speedup	1	1.94	3.93	4.42	6.16	8.86	8.35	8.85	9.00	8.99	**15.86**
n=5e6	Time	39.97	20.63	10.78	8.33	6.53	4.49	4.66	4.71	4.60	4.36	**2.75**
	Speedup	1	1.93	3.70	4.79	6.11	8.89	8.56	8.47	8.68	9.15	**14.48**

Table 1: OpenMP Blocked Matrix Multiplication, 20x20 number of blocks

. . . .

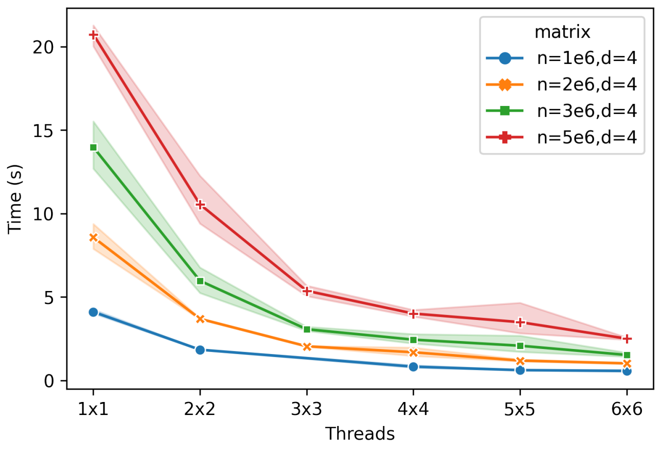


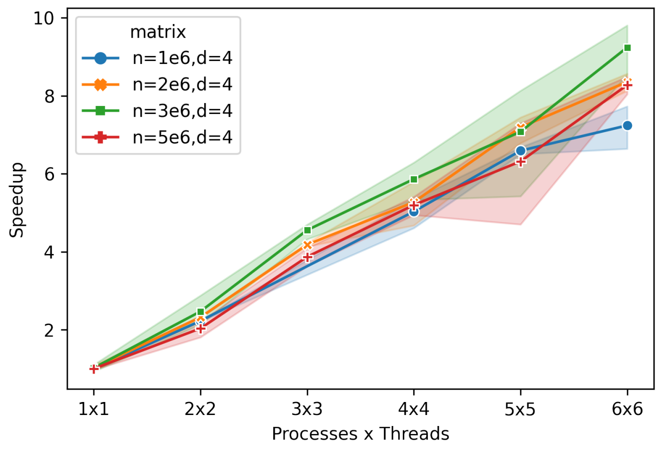


[1]The graphs for the hybrid versions stops at 6x6 although 8x8 was expected to be included. Some TCP connections errors happened when we used more than one node in the HPC losing scheduled jobs

Processes		1	2	4	6	8	10	12	14	16	18	20
n=1e6	Time (s)	9.21	4.88	2.74	1.95	1.49	1.40	1.11	1.09	1.05	1.07	**0.99**
	Speedup	1	1.88	3.35	4.70	6.14	6.57	8.23	8.41	8.70	8.57	**9.27**
n=2e6	Time (s)	18.77	10.03	5.54	3.93	2.96	2.82	2.28	2.20	2.15	2.14	**1.98**
	Speedup	1	1.87	3.38	4.47	6.32	6.64	8.21	8.50	.871	8.76	**9.48**
n=3e6	Time (s)	28.16	14.84	8.16	5,93	4.45	4.11	3.35	3.33	3.17	3.15	**2.94**
	Speedup	1	1.89	3.45	4.74	6.32	6.85	8.39	8.43	8.85	8.94	**9.55**
n=5e6	Time (s)	46.47	24.84	13.95	10.04	7.65	7.11	5.70	5.56	5.35	5.59	**5.07**
	Speedup	1	1.87	3.33	4.62	6.06	6.53	8.14	8.34	8.67	8.31	**9.15**

Table 2: MPI Blocked Matrix Multiplication, 20x20 number of blocks





Processes x Threads		1x1	2x2	3x3	4x4	5x5	6x6
n=1e6	Time (s)	4.11	1.84	1.4	0.82	0.62	**0.56**
	Speedup	1	2.23	3	5	6.6	**7.1**
n=2e6	Time (s)	8.58	3.9	2.04	1.6	1.2	**1**
	Speedup	1	2.32	4.18	5.06	7.15	**8.34**
n=3e6	Time (s)	13.98	6	3.06	2.44	2.08	**1.5**
	Speedup	1	2.34	4.54	5.7	6.52	**9.10**
n=5e6	Time (s)	20.73	10.54	5.36	4.00	3.4	**1.25**
	Speedup	1	2	3.85	5.16	5.9	**8.26**

Table 3: Hybrid Blocked Matrix Multiplication, 64x1 number of blocks

3.1 Serial

The number of blocks is a noticeable factor considering performance, thus in the table below, the variability of the serial execution time with respect to the number of blocks (squared) is tested. The time required to convert the input data to A-CSR, B-CSC, F-CSR can be found under the cell `mm2csx` and for each pair of block dimensions, the time required to create a blocked sparse data structure (BSX) from a CSX format is also listed.

	mm2csx	1x1	20x20		30x30		40x40		50x50		60x60	
			BSX	BMM	BSX	BMM	BSX	BMM	BSX	BMM	BSX	BMM
n=1e6	2.80	**0.37**	1.39	7.92	2.04	15.80	2.64	24.133	3.28	36.51	3.94	52.78
n=2e6	6.41	**0.73**	2.77	16.18	4.20	32.62	5.41	75.27	7.84	75.27	9.31	106.92
n=3e6	9.90	**1.35**	4.04	23.87	7.46	47.91	9.00	112.31	13.12	112.31	15.02	156.02
n=5e6	14.71	**2.11**	6.81	38.72	12.67	83.05	16.32	182.20	19.73	182.20	26.44	259.95

Table 4: Serial execution times (s) with respect the number of number of blocks
CSX -> CSR or CSC, BSX -> BSR or BSC

4 Conclusions

4.1 Blocking

Due to the decision of the CSR structure to the inner level of the block matrix multiplication, two significant bottlenecks are induced that result to poor performance when scaling the number of blocks. These bottlenecks are the functions `updateC()` and `updateMask()` (Algorithm 2).

Maintaining and updating a CSR structure isn't performance friendly because there is the serialized dependency of tracking the non zero elements per row. Thus for each update, we need to modify the whole pointer array while setting the column values to the valid indices of the non zero elements array. This comes with a computing cost because almost all the elements of the array are moved within the memory. In this version, instead of keeping the same object and resizing and moving the elements, for readability reasons, we created a new object per update, which comes with significant cost too. The same, of course, applies with the CSC structure.

To solve the aforementioned issue, a more convenient data structure to store the result of the inner block matrix multiplication would be the COO which comes with a memory trade-off.

For the non-blocking serial version (1x1), it should be noted that we have very good performance and it exceeds the corresponding `Matlab`'s implementation.

4.2 Parallelism

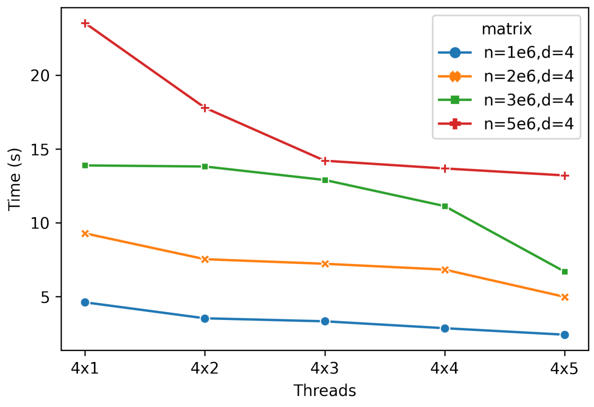
Although the implementation is heavily dependent with the number of blocks created as a reference point, **embarrassingly parallel** performance can be observed. This can happen, because each task is responsible for a set of rows of the final matrix product which is independent among threads. Thus, no communication overhead and synchronization between tasks is required. Indeed, the speedup graphs show a strong linear relationship as we increase the number of tasks. The characteristics of the matrices, the way they are created, are suitable for load sharing with respect to the rows, because there won't be many zero blocks. Thus each task will have sufficient work to do.

Comparing OpenMP and OpenMPI performance, we can conclude that shared memory applications with the same number of tasks will have better performance than the distributed one. This is reasonable because there is no communication overhead. In this version, the communication overhead is induced when aggregating the results of each process. On the other hand, shared memory applications can cause false sharing due to cache coherency if work scheduling isn't treated carefully. This can't happen to independent processes without multithreading since each process has its own CPU core and memory.

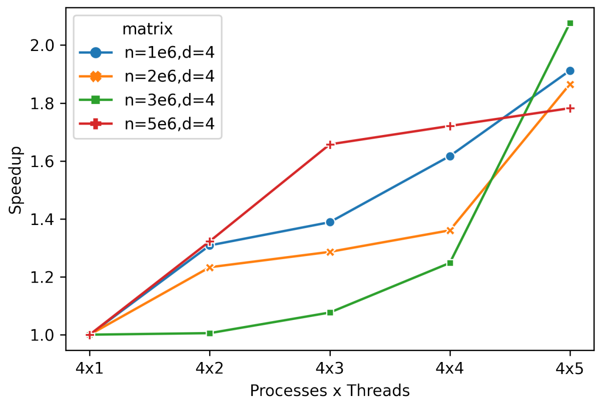
Regarding the hybrid version (OpenMPI/OpenMP), comparing the total number of tasks with the equivalent number of threads and processes for the OpenMP and OpenMPI respectively, the performance doesn't seem to improve. In other words, the number of tasks that we request for the hybrid version could perform better using only one parallel framework. In order to illustrate this we test a specific case of 4 processors and a varying number of threads.

So we have the pairs 4x1=4, 4x2=8, 4x3=12, 4x4=16, 4x5=20. For this experiment we set blocks dimensions 20x20, to have a common reference point with respect to the number of blocks between the OpenMP, OpenMPI and Hybrid versions. As we can see below, the hybrid one, having a fixed number of processes and increasing the number of threads, doesn't scale better than the other parallel frameworks on their own for the equivalent number of tasks.

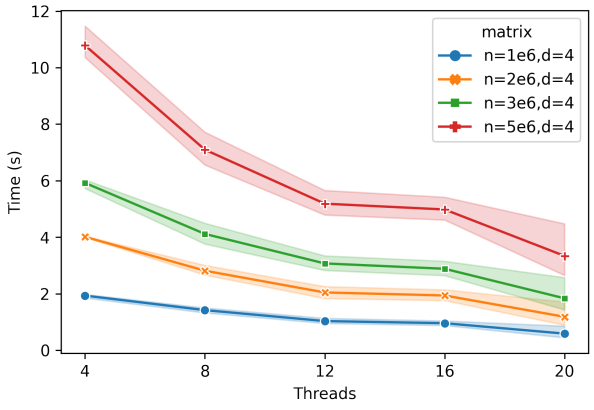
One reason on why this might happening, although it isn't clear personally speaking, is the induced overhead of two run time systems.



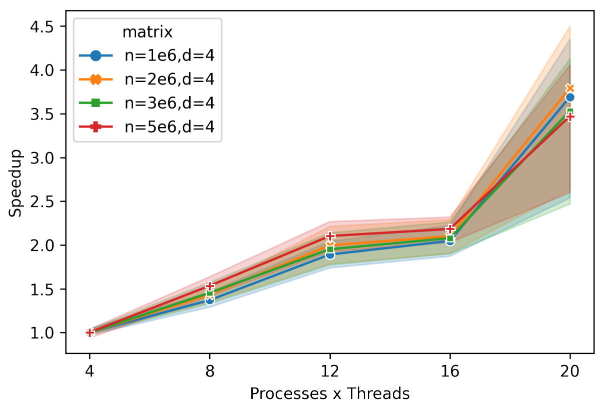
(a) Hybrid Time(s)



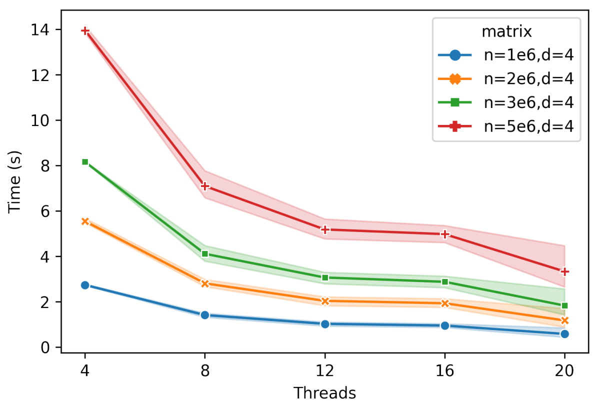
(b) Hybrid Speedup



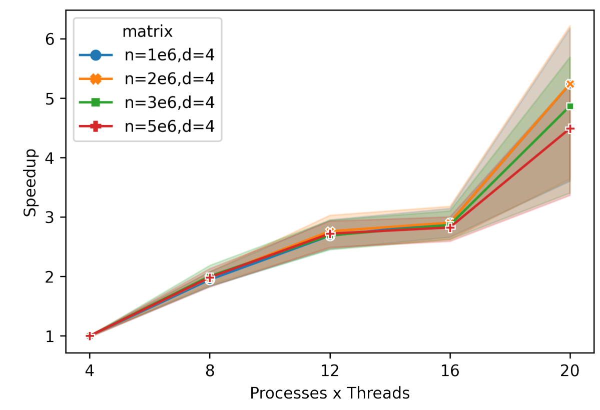
(c) OpenMP Time(s)



(d) OpenMP Speedup



(e) OpenMPI Time(s)



(f) OpenMPI Speedup

Figure 4: Comparison of versions with respect to the same number of tasks with reference point 4 tasks

5 Validation

For validation and testing, `Python` scripts were used with the `SciPy` library for sparse matrices calculations. In the first stage, random sparse matrices are created. In the second, our implementation is executed and finally via I/O files, the results are compared. A bash script is provided for the aforementioned.

6 Repository

The implementation of the aforementioned can be found at this `Github` repository. Aristotelis' HPC was used for the experiments.

References

[1] Gregory Bard. Achieving a log(n) speed up for boolean matrix operations and calculating the complexity of the dense linear algebra step of algebraic stream ciper attacks and of integer factorization methods. *IACR Cryptology ePrint Archive*, 2006:163, 01 2006.

[2] Uriel Feige. Fast matrix multiplication. `https://www.wisdom.weizmann.ac.il/~feige/Algs2014/lecture11FastMM.pdf`.