(0, 1)-Matrix-Vector Product Computations via Minimum Spanning Trees

Jeffrey D. Witthuhn
Computer Science Department
Saint Cloud State University
Saint Cloud, MN, 56301
jeffdwitt@gmail.com

Andrew A. Anda
Computer Science Department
Saint Cloud State University
Saint Cloud, MN, 56301
aanda@stcloudstate.edu

Abstract

A (0, 1) matrix is a matrix where all of its elements are from the set {0, 1}. The complexity of a matrix-vector product is $O(n^2)$, and although there is no known complexity reducing method for a matrix-vector product, our algorithm seeks to significantly reduce the total number of operations by exploiting redundancy in a (0,1) matrix-vector product. The key observation is that since each element in the result vector is a sum of some elements in the operand vector, we can calculate each element in the result vector by using previously computed elements plus a sum of elements in the operand vector multiplied by ±1 where the corresponding rows of the (0,1) matrix differ. Our algorithm uses this observation by generating a complete graph, where the rows in the (0, 1) matrix comprise the vertices, and the Hamming distances between rows comprise the edge weights. Next, a minimum spanning tree (MST) of this graph is generated which yields the minimum number of arithmetic operations for the matrix-vector product when traversed. To calculate the product, we first choose a start vertex, calculate the first result matrix element, and then perform a traversal of the tree where visiting a vertex is calculating a new result matrix element based on the previously calculated element. We also vertically partitioned the matrix to further reduce the number of operations via a set of MSTs having significantly lower total spanning weights. We plot the ratio of the arithmetic operations count vs the following variables: ratio of rows and columns, the number of elements in the matrix, the sparsity ratio, and the number of partitions. Our results show that as the ratio of rows to columns increase the amount of operations required decreases, when the size of the matrix increases the ratio of the operation counts approaches 1, the amount of operations required decreases with the percentage of 1's in the matrix, and the number of operations required decreases with the number of partitions. We conclude that this method can be useful in cases where doing the MST calculation is not too costly, such as when reusing the same matrix for many products, and the matrix is not too sparse; the benefits increase with more partitions of the matrix and with smaller matrices.

1 Introduction

A (0, 1)-matrix is any matrix where all of the elements are from the set {0, 1}, these arise from computational problems in a variety of application areas including computational graph theory and information retrieval. To multiply (0, 1) matrices, because each non-zero element of the matrix is 1, simply perform a sum of the operand vector elements corresponding to the non-zero elements of the respective row of the matrix is all that's needed. The complexity of the general product of two matrices is $O(n^3)$, which can be reduced to $O(n^{\lg 7})$ with Strassen's method Although there is no known comparable complexity-reducing algorithm that can be applied to the matrix-vector product which is $O(n^2)$. The purpose of our experiment is to reduce the total number of operations using an observation that allows us to use previously calculated elements in the result vector to get the rest of the elements by exploiting redundancy in the calculation of the (0,1) matrix-vector product and allowing us to only do an operation when the two respective rows differ. We exploit this observation by creating a minimum spanning tree (MST) over the complete graph wherein the matrix rows comprise the set of vertices connected by edges in which the weights represent the Hamming distances between each pair of rows. This was then implemented in C++ using first what proved to be an inefficient implementation followed by refactoring to a more efficient implementation exploiting the BOOST graph library. We compare our new method with the conventional method.

2 Key Observation

The conventional method for calculating the (0, 1) matrix-vector product requires a doubly nested loop and has time complexity $O(MN)$, where M is the number of rows and N is the number of columns (Figure 2.1).

$$(A_{M\times N} * X_{N\times 1} == B_{M\times 1})$$

```
//B is initialized to zero
for (int i=0; i < M; i++)
    for (int j=0; j < N; j++)
    {
        if (A[i][j])//A[i][j]==1
            B[i]+=X[j];
    }
```

Figure 2.1: C++ style pseudo code for (0, 1) matrix-vector product.

Notice that each B_i is a sum of all X_j where A_{ij} is equal to 1. Also, if we know some B_x, then any B_y can be calculated from B_x (Figure 2.2).

```
B[y]=B[x];
for (int j=0; j < N; j++)
    {
    if (A[x][j] ^ A[y][j])//Axj XOR Ayj
        {
        if (A[x][j]==1)
            B[y]-=X[j];
        else //A[x][j]==0
            B[y]+=X[j];
        }
    }
```

Figure 2.2: Calculating B_y from B_x:
(1) Assign B_x to B_y.
(2) For each value that A_{xj} and A_{yj} differ, either subtract or add X_j to or from B_y.

Observe that the number of operations in the calculation of B_y is the number of bits that row A_x differs from row A_y, that is the number of operations to calculate B_y from B_x is the Hamming distance between the row of bits A_x and A_y. We use this observation to reduce the number of arithmetic operations required to compute all B_i by applying graph theory to this problem.

3 Multiplication Using a Minimum Spanning Tree

Because we can get a new B_y from B_x in an amount of operations equal to the Hamming distance of two rows, to minimize the amount of operations, we want to find which rows to use to calculate other rows. That is, which rows have the least Hamming distance between each other? To do this we can create a complete weighted graph where each row represents a vertex and the Hamming distance between each vertex represents the weight of each edge. We then can find the MST of this graph to obtain a structure that when traversed will result in the least amount of operations with our method (Figure 3.1). Once we have computed the MST, we pick a starting vertex and calculate its corresponding result matrix entry explicitly and conventionally, then calculate the rest of the result matrix entries from the first by traversing the graph from that starting vertex. When multiplication is carried out by traversing the MST, it will result in the minimum amount of total additions and subtractions for this method (without adding spurious vertices). We observed that the (0, 1) matrix can be partitioned vertically so that the matrix-vector product is decomposed into multiple partial matrix-vector products summed together to yield the same result vector. Choosing a starting point could also be important, and two main options were considered: choosing a most-connected vertex, or the sparsest vertex. We chose the sparsest vertex because it would save the most operations (e.g. applying our differencing method sequentially to the matrix in Figure 3.1 from the bottom row up results in fewer operations than from the top row down), but starting from a most-connected vertex, or a graph center of the MST, could reduce the total traversal time if performed in parallel.

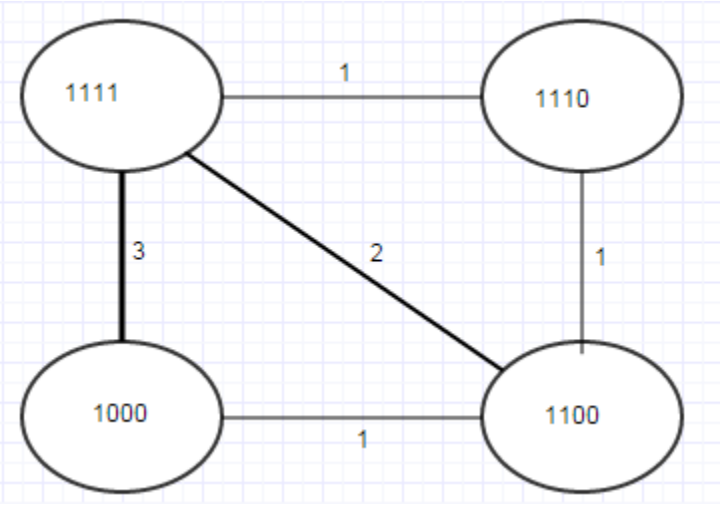

Figure 3.1: Graph of the (0, 1) matrix: $\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$

4 Implementation

We implemented our algorithm in C++ with the standard library and with the BOOST library. Our implementation is within 2 classes. First the `Bitmatrix` class is the class where most of the operations are carried out, its main data structure is a vector of dynamic bitsets which is defined in the BOOST library, for our purposes they are sets containing only 0's or 1's that can be accessed like arrays and operations can be carried out on them like binary numbers, e.g. `^` (`XOR`). The `Bitmatrix` class has all of the methods for calculating its own MST and the MST does not need to be calculated more than once if the members of the instance of `Bitmatrix` do not change. It also has the methods for multiplying both normally and via MST itself with an object of the `Doublematrix` type. The second class, `Doublematrix`, whose main data structure is a `vector` of `vector<double>`, for the purpose of this experiment though the second dimension of `Doublematrix` is always 1. Both classes have methods to allow partitioning of the problem, randomizing themselves based off of a seed from `std::chrono`, and getting/setting elements along with other methods for testing purposes. The MST calculation was at first implemented using naive and inefficient methods which made calculating any matrix with more than 50 seem unreasonable, possibly due to many vector operations for finding minimum edge. So the BOOST graph library was then used to have a quick and efficient way to find the MST. The original implementation of the graph traversal seemed to work fine however so that was maintained.

```cpp
#include "bitmatrix.h"
#include <fstream>
//This program multiplies a random 4x4 boolean(0, 1)
//matrix with a 4x1 matrix of values of type double
int main()
{
    Bitmatrix matrix4x4(4,4); //initialize the 01 matrix
    Doublematrix vector4x1(4,1);//initialize the operand vector
    Doublematrix resultvector1(4,1);//initialize the first result vector
    Doublematrix resultvector2(4,1);//initialize the second result vector
    matrix4x4.randomize(); //randomizes the 01 matrix using boost::random
    vector4x1.randomize(1,50);//randomizes the vector between 1 and 50 using boost::random
    cout<<"(0,1) matrix:\n";
    matrix4x4.rprint();//prints the 01 matrix
    cout<<"column vector:\n";
    vector4x1.rprint();//prints the vector
    cout<<"operations required normally: ";
    resultvector1=matrix4x4.mult(vector4x1,cout);//normally multiply the 01 matrix by the vector
    //then stores it in result vector, it also prints the number of operations to ostream parameter
    cout<<"\nresult vector 1:\n";
    resultvector1.rprint();//prints the result vector
    cout<<"operations required with MST: ";
    matrix4x4.boostcalcmst();//calculates the MST using boost graph library
    matrix4x4.boostcalcalist();//fills an adjacency list form the calculated MST
    resultvector2=matrix4x4.multmst(vector4x1,cout);//uses the MST to multiply the 01 matrix by the vector
    //then stores it in result vector, it also prints the number of operations to ostream parameter
    cout<<"\nresult vector 2:\n";
    resultvector2.rprint();//prints the result vector
    return 0;
}
```

Figure 4.1: C++ code for the calculation of the (0, 1) matrix-vector product of a random (0, 1) matrix and a random vector of doubles

```
(0,1) matrix:
0100
1101
1001
0111
column vector:
13
27
19
15
operations required normally: 9
result vector 1:
27
55
28
61
operations required with MST: 5
result vector 2:
27
55
28
61
```

Figure 4.2: An execution of the code from Figure 4.1

After implementing the multiplication using the MST method, the next step was to proceed to implement the partitioning of the (0, 1) matrix into a specified number of vertical partitions, then generate and traverse the MST for each partition. We then multiply each of those partitions by the operand vector and sum all partial result vectors to equal the final result vector. This decomposition should result in even fewer operations

because finding the MST of each section will allows each section to be calculated in a different order and the sum of all of the weights of all calculated MSTs will be smaller than if it was all done in one section.

```
#include "bitmatrix.h"
#include <fstream>
//This program creates a ramdom 1024x1014 binary (0,1) matrix and
//multiplies it by a random 1014x1 vector of doubles.
//it compares the amount of operations required to multiply it
//normally, via the MST method, and via the MST method with 16
//partitions.
int main()
{
    int numberOfPartitions=16;
    int rows=1024;
    int columns=1024;
    Bitmatrix matrix4x4(rows,columns);//initialize the 01 matrix
    Doublematrix operandVector(columns,1);//initialize the operand vector
    Doublematrix resultvector1(rows,1);//initialize the first result vector
    Doublematrix resultvector2(rows,1);//initialize the second result vector
    Doublematrix resultvector3(rows,1);//initialize the third result vector
    matrix4x4.randomize(); //randomizes the 01 matrix using boost::random
    operandVector.randomize(1,50);//randomizes the vector between 1 and 50 using boost::random
    cout<<"\n   operations required normally: ";
    resultvector1=matrix4x4.mult(operandVector,cout);//normally multiply the 01 matrix by the vector
    //then stores it in result vector, it also prints the number of operations to ostream parameter
    cout<<"\n   operations required with original MST method (1 partition): ";
    matrix4x4.boostcalcmst();//calculates the MST using boost graph library
    matrix4x4.boostcalcalist();//fills an adjacency list from the calculated MST
    resultvector2=matrix4x4.multmst(operandVector,cout);//uses the MST to multiply the 01 matrix by the vector
    //then stores it in result vector, it also prints the number of operations to ostream parameter
    cout<<"\n   operations required with MST partitioned into 16 vertical sections: ";
    resultvector3=matrix4x4.multmstpartitioned(operandVector,cout,numberOfPartitions);//splits the (0,1) matrix into
    //numberOfPartitions partitions, calculates the MST of each of them, multiplies each partition into a
    //partial result matrix, then sums all partial result matrices into the result matrix and returns.
    return 0;
}
```

Figure 4.3: code that multiplies a random 1024x1024 (0, 1) matrix by a 1024x1 vector and compares the conventional method, our MST method (1 partition), and our MST method with 16 partitions of 64 bits.

```
operations required normally: 524528
operations required with original MST method (1 partition): 472568
operations required with MST partitioned into 16 vertical sections: 320130
```

Figure 4.4: an execution of the program in Figure 4.3

There are many improvements to the implementation that can be made for efficiency and utility, however the current implementation is sufficient for counting and comparing the amount of additions and subtractions required by the method. In the next sections we discuss our testing of our implementation.

5 Tests and Results.

There are many different factors which we can vary for these tests. The factors which we decided were the most important to change and observe were: the ratio of the number of rows to the number of columns in the (0, 1) matrix, number of elements in the (0, 1) matrix, the sparsity of the matrix, and the number of partitions we have in the matrix. We also computed the reduction in the number of operations when calculating the product using an example (0, 1) matrix from the web.

5.1 Rows to Columns Ratio

Testing how the ratio of rows to columns variable changes the operation count is challenging because we do not want to alter the total number of elements in the matrix, we will have a sparsity of about 50%, and look at both 1 partition and 4 partitions. To do this with only altering the ratio variable, we pick a number for the total amount of elements to keep constant, we picked 1024^2. Then we find all of the factors of 1024^2 and divide 1024^2 by some of those factors to get the test row/column combinations with a constant total number of elements, this allows only allows for 10 data points though because of some memory limitations with having a ratio or rows to columns greater than 16. Finally we can plot $\frac{numberOfOperationsWithMST}{numberOfOperationsWithBruteForce}$ as a function of $\frac{rows}{columns}$ (see Figure/Table 5.1).

$\frac{rows}{columns}$	$\frac{MSTmethod}{BruteForce}$	$\frac{4Partitions}{BruteForce}$
16	0.780369007	no data
4	0.852026759	0.705711636
1	0.902327455	0.803874064
0.25	0.932282729	0.868411318
0.0625	0.953266451	0.910275245
0.015625	0.964031067	0.937785014
0.00390625	0.968554901	0.952112133
0.000976563	0.957257828	0.947084566
0.000244141	0.931229018	0.925705619
6.10E-05	0.873104178	0.871485468

Table 5.1:
Row 1: ratio of rows to columns
Row 2: ratio of number of operations using our method vs the conventional brute force method
Row 3: ratio of number of operations using our method with 4 partitions vs the conventional brute force method

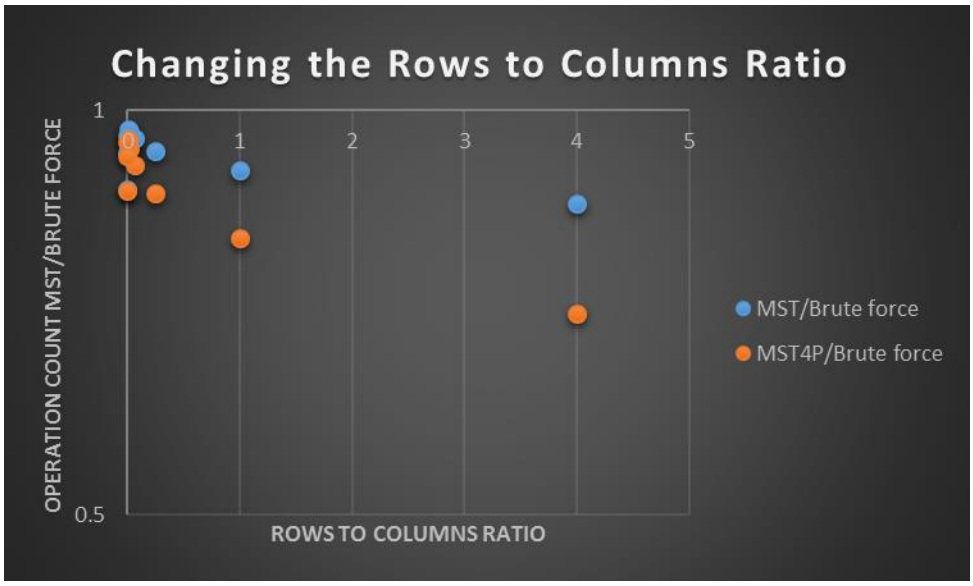

Figure 5.1: plotting the data in Table 5.1
Y axis = operations brute force/operations MST method
X axis = rows/columns ratio
Smaller *Y* axis values mean that more operations were eliminated.

From the data we can infer that when roughly $0<X<0.01$, the operations ratio is increasing, and that for roughly $0.01<X$ the operations ratio is decreasing.

5.2 Number of Elements

Testing how the number of elements changes the operation count is not as challenging as the previous. In this test we use square matrices for simplicity and to have a constant rows/columns ratio of 1, a sparsity of about 50%, both 1 partition and 4 partitions and will test number of element values from 100^2 to number of 3300^2. (See Table 5.2 and Figure 5.2)

#of elements	$\frac{MSTmethod}{BruteForce}$	$\frac{4Partitions}{BruteForce}$
10000	0.742139608	0.520664426
40000	0.810282622	0.631990379
90000	0.833676656	0.676811305
160000	0.856069884	0.714026844
250000	0.869955732	0.740192147
360000	0.874453713	0.754026444
490000	0.884548615	0.770539146
640000	0.890365085	0.782196638
810000	0.896334923	0.79381163
1.00E+06	0.898251081	0.800371558

#of elements	$\frac{MSTmethod}{BruteForce}$	$\frac{4Partitions}{BruteForce}$
1.21E+06	0.903421281	0.808870736
1.44E+06	0.906712416	0.814398832
1.69E+06	0.909484609	0.820753077
1.96E+06	0.912406257	0.826283427
2.25E+06	0.91509444	0.831481024
2.56E+06	0.91751214	0.835872553
2.89E+06	0.920218104	0.840987493
3.24E+06	0.921630353	0.843637926
3.61E+06	0.924058733	0.848058688
4.00E+06	0.924177213	0.849844702
4.41E+06	0.926943967	0.854050392
4.84E+06	0.928026019	0.856211967
5.29E+06	0.928638694	0.858510074
5.76E+06	0.929943953	0.860977813
6.25E+06	0.931609598	0.863488573
6.76E+06	0.933320209	0.866596602
7.29E+06	0.933745092	0.867997155
7.84E+06	0.934232611	0.869567967
8.41E+06	0.935713326	0.871778528
9.00E+06	0.93668265	0.873693112
9.61E+06	0.937576237	0.875572669
1.02E+07	0.938423011	0.877209889
1.09E+07	0.938891664	0.878350591

Table 5.2: Number of rows, operations ratio MST, Operations ratio with 4 partitions.

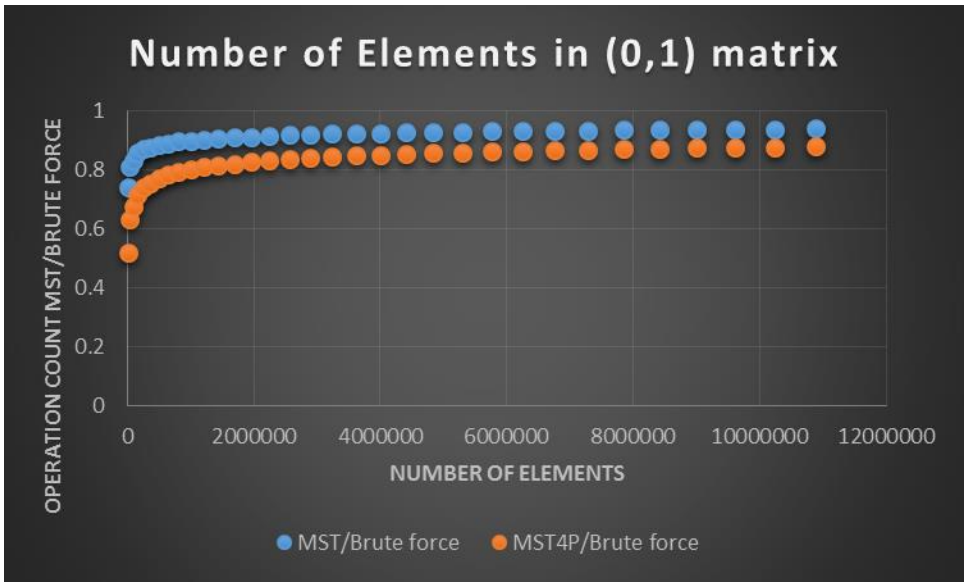

Figure 5.2: Plot of operation ratios vs number of elements in the matrix.

From figure 5.2 we can infer that as the number of elements in the matrix increases, the benefit from this method decreases. Determining when or if it exceeds an operation count ratio of 1 will require further analysis.

5.3 Sparsity

In testing how the sparsity affects the operation count, we once again keep the ratio of columns and rows constant by using square matrices, we keep the number of elements constant by using a matrix with 1024^2 elements, and we once again test using both 1 and 4 partitions. We will plot the operation ratios vs the percentage of 1's in the (0, 1) matrix. (See Table 5.3 and Figure 5.3).

percentage of 1's	$\frac{MSTmethod}{BruteForce}$	$\frac{4Partitions}{BruteForce}$
1	1.084279	0.710795
6	1.506526	1.172866
11	1.501967	1.235128
16	1.453676	1.23941
21	1.385651	1.200023
26	1.309967	1.145476
31	1.23142	1.084161
36	1.146971	1.016007
41	1.061787	0.944509
46	0.973008	0.866736
51	0.883702	0.788008
56	0.792503	0.705673
61	0.700905	0.622278
66	0.607748	0.537372
71	0.5166	0.453451
76	0.424506	0.369967
81	0.331079	0.285644
86	0.237376	0.200962
91	0.149274	0.122126
96	0.06078	0.045277

Table 5.3: percentage of 1's in the (0, 1) matrix, Operation ratio with MST, Operation ratio for MST with 4 partitions.

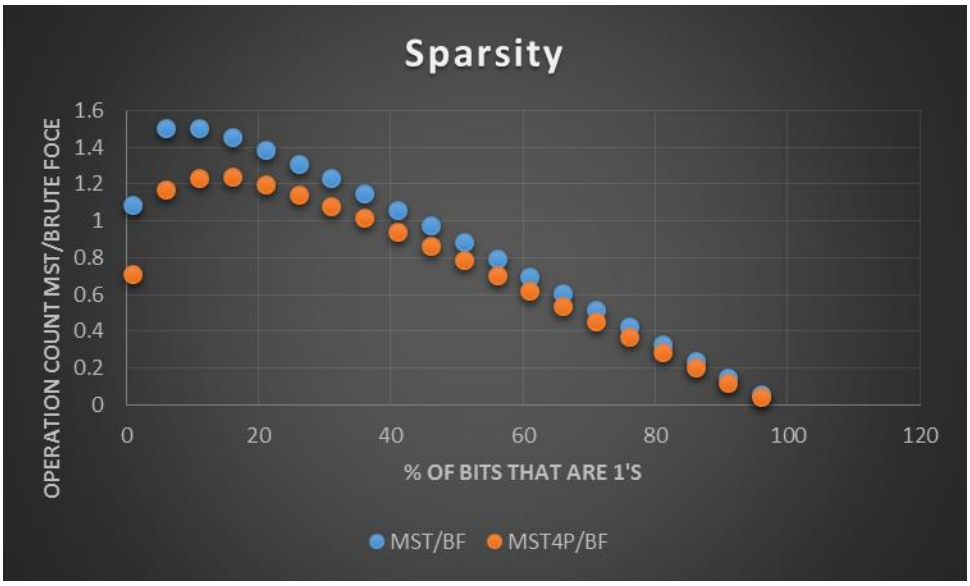

Figure 5.3: Plot of number of operations vs the % of 1's in the (0, 1) matrix.

Figure 5.3 illustrates that with very sparse matrices this method is not effective and has up to 50% more operations required. However as the number of 1's in the matrix increases the amount of operations drops dramatically. At 100% there would be additions equal to the number of columns, and then there would be no further additions.

5.4 Number of Partitions

To test how the number of partitions effects the operation count we decided to once again keep the ratio of rows/columns constant by using a square matrix, keep the number of elements in the matrix constant by fixing it at 1024^2 and having a sparsity of around 50%. We will plot the operation ratio vs the number of partitions for 2, 4, 8, 12, 16 and 32 partitions (See Figure 5.4, and Table 5.4).

Partitions	$\frac{MSTmethod}{BruteForce}$
1	0.90202
2	0.861673
4	0.803549
8	0.722614
16	0.611066
32	0.457676

Table 5.4: number of partitions and operation ratio

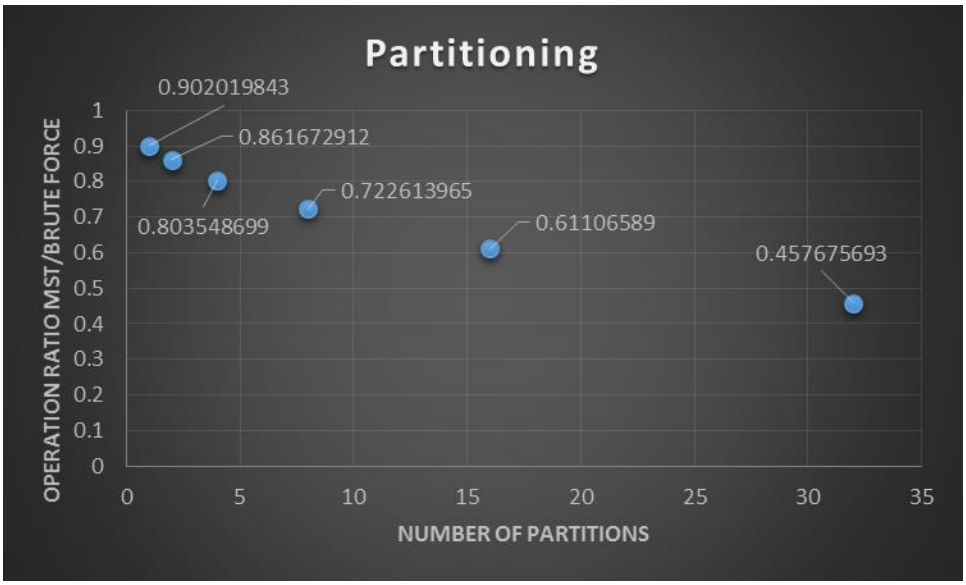

Figure 5.4: Operation ratio vs number of partitions

From Figure 5.4 we can see that as the number of partitions increase, the operation ratio decreases. The amount of operations decrease as the number of partitions increase.

5.5 Matrix Example

This section is inteded to demonstrate this method using a non-random matrix. We use the following term-document matrix for our example:

Document-Term Matrix

	t1	t2	t3	t4	t5	t6
D1	24	21	9	0	0	3
D2	32	10	5	0	3	0
D3	12	16	5	0	0	0
D4	6	7	2	0	0	0
D5	43	31	20	0	3	0
D6	2	0	0	18	7	16
D7	0	0	1	32	12	0
D8	3	0	0	22	4	2
D9	1	0	0	34	27	25
D10	6	0	0	17	4	23

Figure 5.5: Term-Document Matrix,
Credit: Sargur N. Srihari, University of Buffalo, NY [3]

We will use this 10×6 matrix and convert it into a (0, 1) matrix where all non-zero elements are 1. And then do our final test where we multiply this matrix by an arbitrary vector and see the results.

	operations	
normal	MST	2 partitions
37	8	9

Table 5.5: Term-Document Matrix multiplication results

MST multiplication using the term document matrix saves 29 operations when compared to the normal method.

6 Future Work

Future work would include:

(1) Refactor the implementation of our algorithm for efficiency - the computation and traversal times of the MST can be reduced.

(2) Refactor our algorithm to implement recently published near-linear complexity algorithms for computing the MST which may exploit the special properties of our graph: complete with positive integer edge weights bounded by the vertex count.

(3) Find where the cutoffs are for the operation savings in regard to sparsity.

(4) Find a way to use the transpose of the matrix when the row/column ratio is less than 1.

(5) Compute each partition in parallel to further reduce computation time.

(6) Test other traversal methods and starting points.

7 Conclusion

We implemented and tested our algorithm for reducing the number of arithmetic operations (additions) required to calculate a (0,1)-matrix-vector product. We found that our algorithm reduces proportionately more operations when the following conditions are satisfied: a lower total number of elements, a greater rows to columns ratio, a greater percentage of 1's and a greater number of partitions. Greater row/column ratios which allow fewer operations help explain why partitioning is effective as well. Our algorithm does reduce operations, however it's usefulness may be limited because of the time overhead required to generate and traverse the minimum spanning tree, and the sparsity and shape of the matrix are also important factors. Our algorithm would be more competitive when reusing the same non-sparse matrix many times (characteristic of Krylov subspace algorithms), and the competitiveness will increase with the number of partitions used with correspondingly narrower matrices.

References

[1] Andrew Anda. *A Gray Code Mediated Data-Oblivious* (0, 1)-*Matrix-Vector Product Algorithm*. Proceedings of the 2005 International Conference on Scientific Computing, CSC 2005, Las Vegas, Nevada, USA, June 20-23, 2005.
[2] Aaron Webb and Andrew Anda. (0, 1)-*Matrix-Vector Products via Compression by Induction of Hierarchical Grammars*", Proceedings of the 38th Midwest Instruction and Computing Symposium (MICS 2005), April 2005.
[3] Image, Sargur N. Srihari, University of Buffalo, NY, http://www.cedar.buffalo.edu/~srihari/CSE626/Lecture-Slides/Ch14-Part2-Text-Retrieval.pdf
[4] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms* (2nd ed.). McGraw-Hill Higher Education. 2001.
[4] Henry S. Warren. *Hacker's delight*. (2nd ed.) Pearson Education, 2012.
[5] Gene H. Golub and Charles F. Van Loan. 1996. *Matrix computations*. 1996. *Johns Hopkins University, Press, Baltimore, MD, USA*: 374-426.
[6] BOOST C++ library, boost.org, 2015.
[7] David Eppstein. ICS 161: *Design and Analysis of Algorithms*; *Lecture notes*: *Minimum Spanning Trees*, 1996, https://www.ics.uci.edu/~eppstein/161/960206.html
[8] Darrell A. Turkington. *Matrix Calculus and Zero-One Matrices*. Cambridge University Press, New York, 2002.