PARALLEL&DISTRUBUTED COMPUTING I – TERM PROJECT

SYMMETRIC FIVE-BANDED MATRIX USING COMPRESS SPARSE ROW FORMAT (CSR) STORAGE SCHEME.

PREPARED BY

NAME	: BAHADIR SÖNMEZ
STUDENT ID	: 702181002
DUE DATE	: 13/01/20

COURSE NAME	: PARALLEL&DISTRUBUTED COMPUTING I
COURSE INSTRUCTOR	: PROF. DR. M. SERDAR ÇELEBİ

1. INTRODUCTION

BLAS operations are routines that give standart building blocks for performing fundamental vector and matrix operations. Level 2 BLAS is defined as matrix – vector operations. But in this project, matrix is sparse. Sparse matrices contain very low amount of non-zero elements. This operations called as Sparse Matrix – Vector Multiplication (SpMV).

This project is about compressing sparse matrix with a Compressed Sparse Row (CSR) format and multiply it with vector in order to get rid of zero elements.

2. METHOD AND ALGORITHM

2.1. Compressed Sparse Row (CSR) Format

CSR format is a format that convert sparse matrix (SM) into three one-dimensional arrays that contain only non-zero values and their informations[1]. It has been used since 1960's.

These arrays are defined as matValues, columnIndices and rowPointers. For example if we have M[8][8] sparse matrix; matValues holds 34 real numbers which are non-zero values in the matrix, columnIndices holds 34 integer numbers that which column has non-zero value at that row and rowPointers holds 9 integer numbers that how many non-zero elements exist until that row.

2.2. CSR Algorithm

Naive dense matrix – vector multiplication algorithm and CSR sparse matrix – vector multiplication algorithm used in this project are seen below (C = Axb).

2.2.1. Naive matrix – vector multiplication:

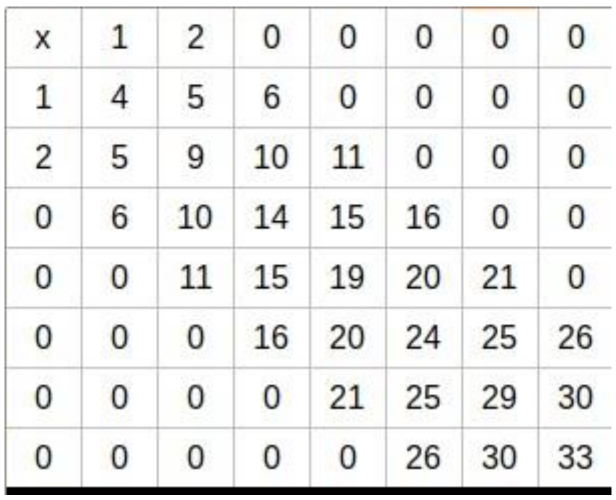
```
for (i=0; i<col; i++) {
   for (j=0; j<row; j++) {
      C[i] += A[j][i] * b[j];
   }
}
```

2.2.2. CSR sparse matrix – vector multiplication:

```
for (i = 0; i < row; i++) {
    C[i] = 0.;
    for (j = rowPointers[i]; j < rowPointers[i+1]; j++) {
        C[i] += matValues[j] * b[columnIndices[j]];
    }
}
```

3. CALCULATION

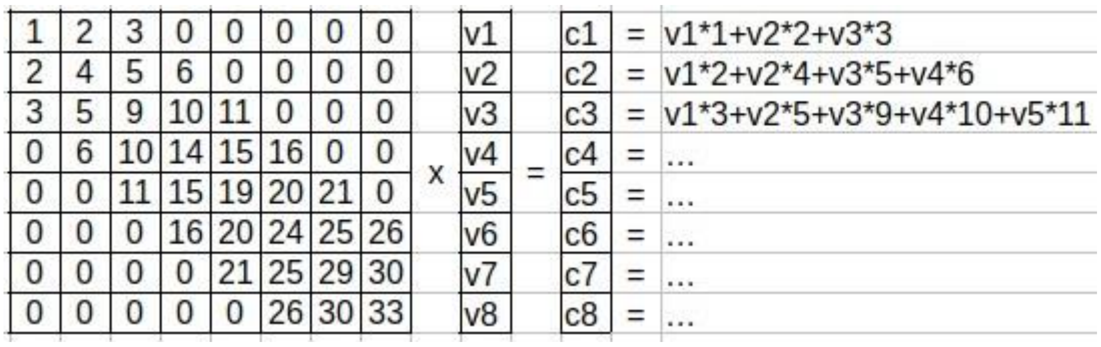
In this project symmetric five-banded matrix – vector multiplication has been done. Five banded symmetric matrix seems like Figure 1.



x	1	2	0	0	0	0	0
1	4	5	6	0	0	0	0
2	5	9	10	11	0	0	0
0	6	10	14	15	16	0	0
0	0	11	15	19	20	21	0
0	0	0	16	20	24	25	26
0	0	0	0	21	25	29	30
0	0	0	0	0	26	30	33

Figure 1 : Five Banded Symmetric Matrix

Multiplication can be seemed at Figure 2. As seen, not all the vector elements and matrix elements multiplied to produce dot product (c1). This information will become important when tuning and optimization approaches tried to applied. It will save massive amount memory and computation time.



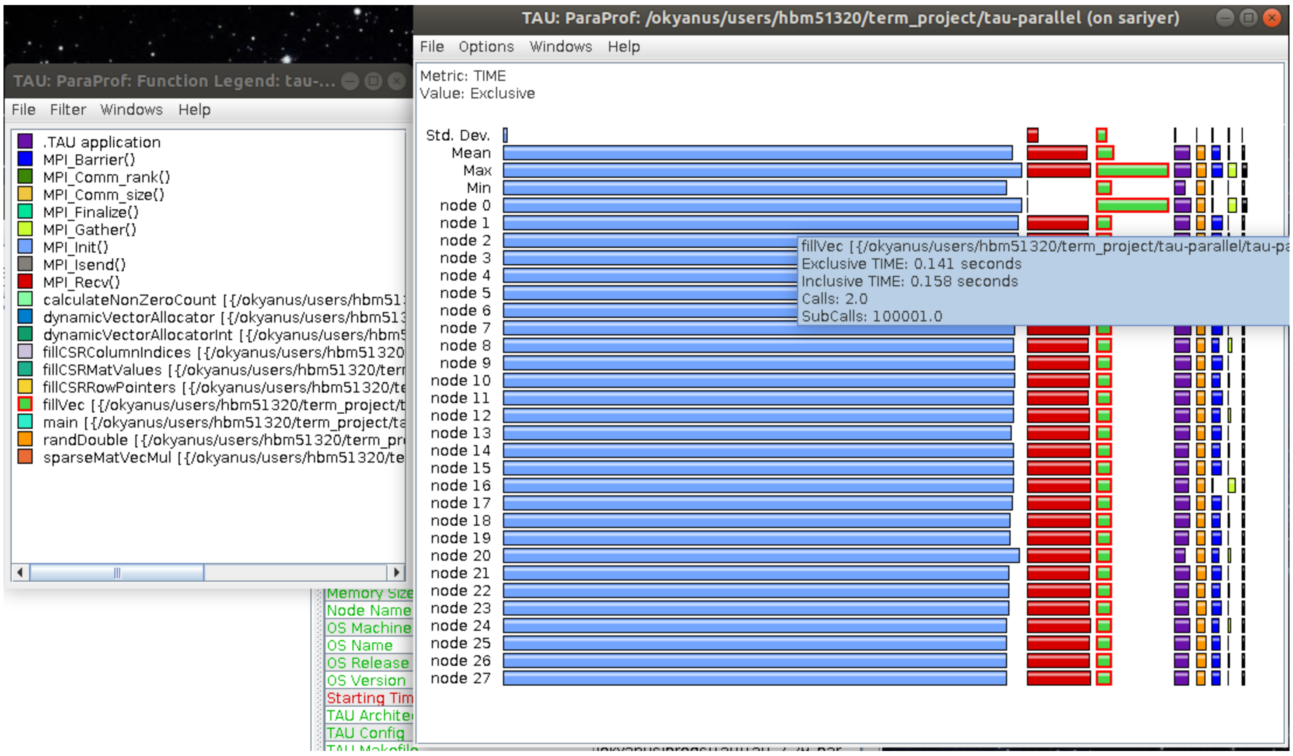
1	2	3	0	0	0	0	0		v1		c1	=	v1*1+v2*2+v3*3
2	4	5	6	0	0	0	0		v2		c2	=	v1*2+v2*4+v3*5+v4*6
3	5	9	10	11	0	0	0		v3		c3	=	v1*3+v2*5+v3*9+v4*10+v5*11
0	6	10	14	15	16	0	0	x	v4	=	c4	=	...
0	0	11	15	19	20	21	0		v5		c5	=	...
0	0	0	16	20	24	25	26		v6		c6	=	...
0	0	0	0	21	25	29	30		v7		c7	=	...
0	0	0	0	0	26	30	33		v8		c8	=	...

Figure 2 : Five Banded Symmetric Matrix – Vector Multiplication

3.1. Parallelization and Tuning

It should be good partitioning for to prevent load imbalance and to get processors into sync. In order to do that, processors need to get similar amount of data and work. At first, all processors produced their own matValues (but they had to communicate with each other to catch the symmetry in whole matrix), columnIndices and rowPointers but vector values produced at master processor then distributed into others their parts. As it said before, processors do not need all the vector values. That approach was easy to handle but it caused load imbalance very badly, and it was not memory friendly. The TAU image can be seen at Figure 3. In that situation while master processor was filling the vector others were waiting for receive the vector values and matValues symmetric values.

In order to avoid that idling, algorithm redesigned as all processors produce their own vector part. Then just they were need to produce their own part and share their first two elements with previous processor to replace their last two elements. In this way, these vector parts at processors created a whole vector and a lot of memory has been saved. The TAU image can be seen at Figure 4.

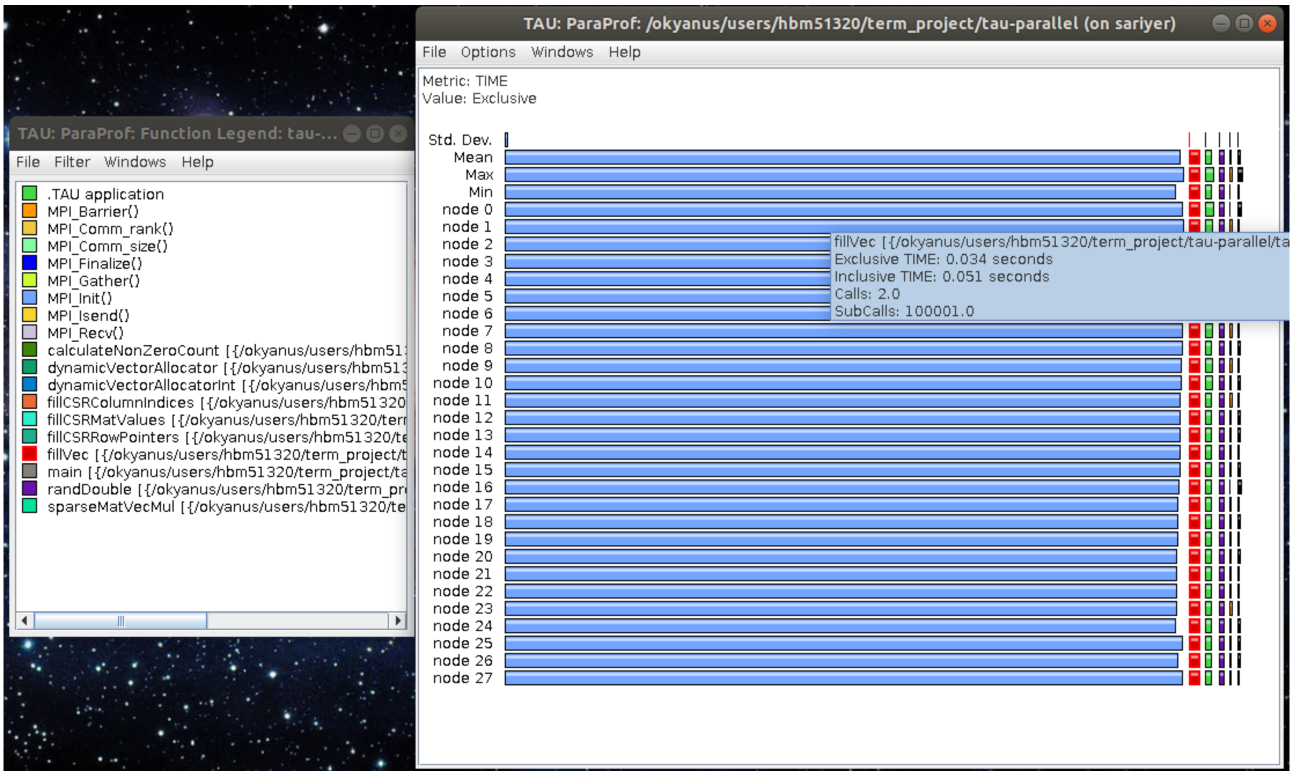


Metric: TIME
Value: Exclusive

🟪 .TAU application
🟦 MPI_Barrier()
🟩 MPI_Comm_rank()
🟨 MPI_Comm_size()
🟩 MPI_Finalize()
🟨 MPI_Gather()
🟦 MPI_Init()
⬛ MPI_Isend()
🟥 MPI_Recv()
🟩 calculateNonZeroCount [{/okyanus/users/hbm51
🟦 dynamicVectorAllocator [{/okyanus/users/hbm51
🟩 dynamicVectorAllocatorInt [{/okyanus/users/hbm5
⬜ fillCSRColumnIndices [{/okyanus/users/hbm51320
🟩 fillCSRMatValues [{/okyanus/users/hbm51320/ter
🟨 fillCSRRowPointers [{/okyanus/users/hbm51320/te
🟥 fillVec [{/okyanus/users/hbm51320/term_project/t
🟦 main [{/okyanus/users/hbm51320/term_project/ta
🟧 randDouble [{/okyanus/users/hbm51320/term_pr
🟧 sparseMatVecMul [{/okyanus/users/hbm51320/te

fillVec [{/okyanus/users/hbm51320/term_project/tau-parallel/tau-pa
Exclusive TIME: 0.141 seconds
Inclusive TIME: 0.158 seconds
Calls: 2.0
SubCalls: 100001.0

Figure 3 : Vector filling and distributing



Metric: TIME
Value: Exclusive

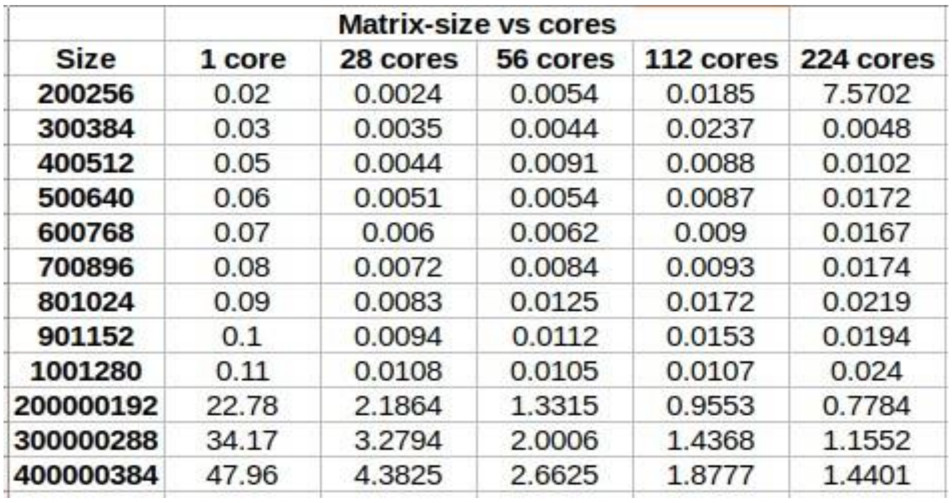
🟩 .TAU application
🟧 MPI_Barrier()
🟧 MPI_Comm_rank()
🟩 MPI_Comm_size()
🟦 MPI_Finalize()
🟨 MPI_Gather()
🟦 MPI_Init()
🟧 MPI_Isend()
⬜ MPI_Recv()
🟩 calculateNonZeroCount [{/okyanus/users/hbm51
🟩 dynamicVectorAllocator [{/okyanus/users/hbm51
🟦 dynamicVectorAllocatorInt [{/okyanus/users/hbm5
🟧 fillCSRColumnIndices [{/okyanus/users/hbm51320
🟩 fillCSRMatValues [{/okyanus/users/hbm51320/ter
🟩 fillCSRRowPointers [{/okyanus/users/hbm51320/te
🟥 fillVec [{/okyanus/users/hbm51320/term_project/t
⬛ main [{/okyanus/users/hbm51320/term_project/ta
🟪 randDouble [{/okyanus/users/hbm51320/term_pr
🟩 sparseMatVecMul [{/okyanus/users/hbm51320/te

fillVec [{/okyanus/users/hbm51320/term_project/tau-parallel/ta
Exclusive TIME: 0.034 seconds
Inclusive TIME: 0.051 seconds
Calls: 2.0
SubCalls: 100001.0

Figure 4 : Vector produced on all the processors

3.2. Testing and Performance

Code has tested to understand performance and speedup. Elapsed time vs. matrix size vs number of cores can be seen at Figure 5. These times shows us higher number of cores are not necessary for this problem, at least not until more bigger sizes.



	Matrix-size vs cores				
Size	1 core	28 cores	56 cores	112 cores	224 cores
200256	0.02	0.0024	0.0054	0.0185	7.5702
300384	0.03	0.0035	0.0044	0.0237	0.0048
400512	0.05	0.0044	0.0091	0.0088	0.0102
500640	0.06	0.0051	0.0054	0.0087	0.0172
600768	0.07	0.006	0.0062	0.009	0.0167
700896	0.08	0.0072	0.0084	0.0093	0.0174
801024	0.09	0.0083	0.0125	0.0172	0.0219
901152	0.1	0.0094	0.0112	0.0153	0.0194
1001280	0.11	0.0108	0.0105	0.0107	0.024
200000192	22.78	2.1864	1.3315	0.9553	0.7784
300000288	34.17	3.2794	2.0006	1.4368	1.1552
400000384	47.96	4.3825	2.6625	1.8777	1.4401

Figure 5 : Elapsed times vs. matrix size vs. number of cores

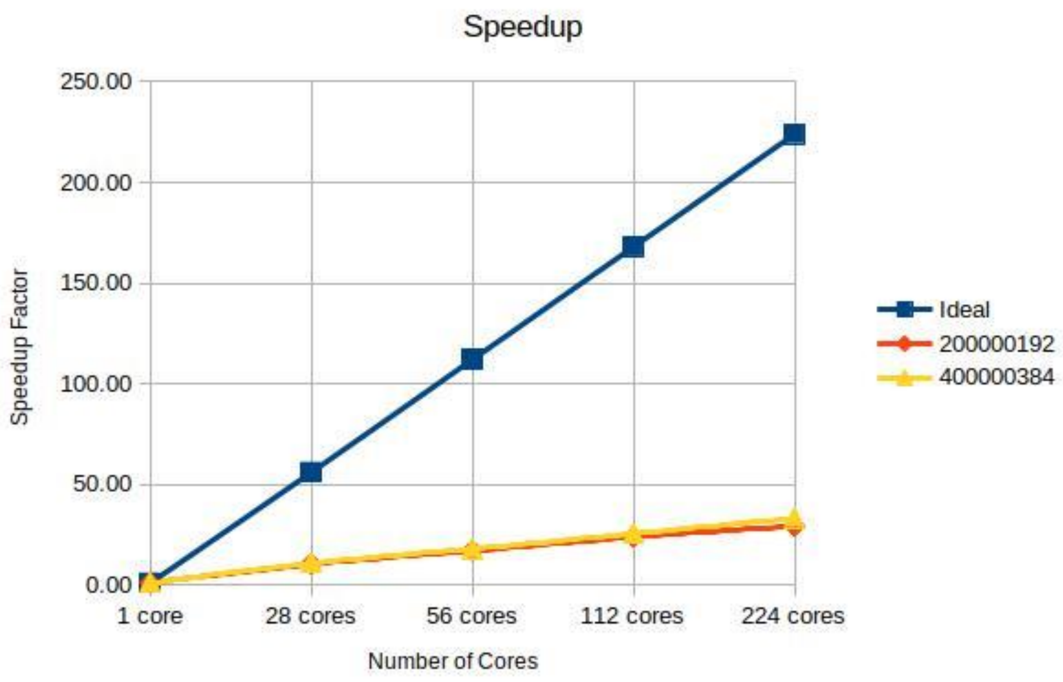


Figure 6 :Speedup factor vs. matrix size

Furthermore these figures at above shows that parallelization overload and serial parts in parallel region effects the speedup badly. These problems and how we overcome them is going to discussed in the results section.

3.2.1. Amdahl’s Law and Speedup

Amdahl’s Law is used for predict theoretical speedup of the execution on parallel programming. Amdahl’s Law can be formulated like below:

$$S_p = \frac{1}{s+(1-\frac{s}{N})} \quad \textbf{(Equation 1)}$$

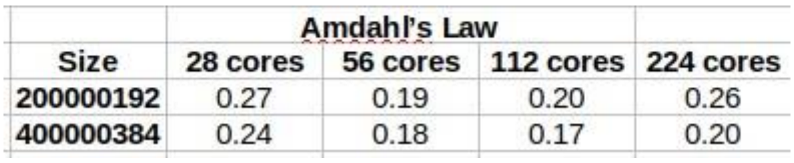
where;

S_p = Speedup

s = Serial fraction

N = Number of cores

We can find our serial parts from the Equation 1. Serial parts can be seen in Figure7.



	Amdahl’s Law			
Size	28 cores	56 cores	112 cores	224 cores
200000192	0.27	0.19	0.20	0.26
400000384	0.24	0.18	0.17	0.20

Figure 7 : Purely serial parts

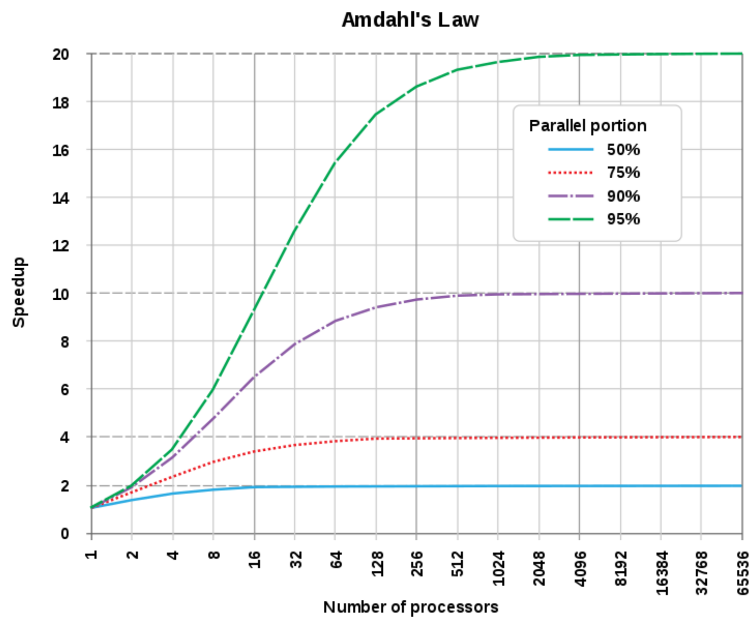
We can calculate parallel portion with the formula by using the values in Figure 7. The algorithm is compared by using general graph of parallel portion formula in Figure 8a and Figure 8b. Parallel portion formula is like that:

$$\frac{1}{(1-p)+\frac{p}{s}} \quad \textbf{(Equation 2)}$$

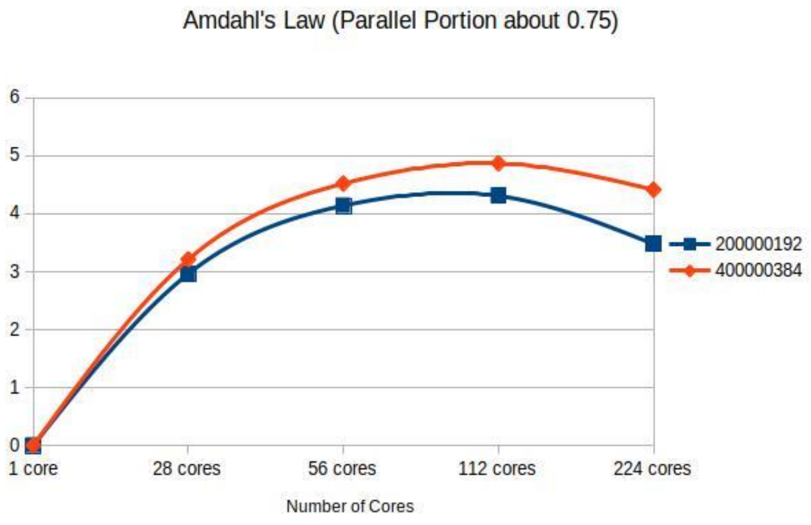
where;

p = Parallel proportion

s = Speedup



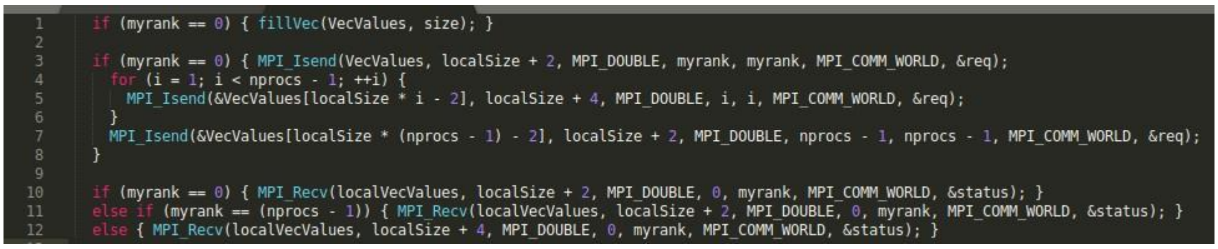
(a)



(b)

Figure 8 : Comparison of general graph and projects graph

Amdahl’s Law also used for improving code performance. In example, two styles of vector partitioning has tried in this project and comparison done between them. First algorithm can be seen at Figure 9.



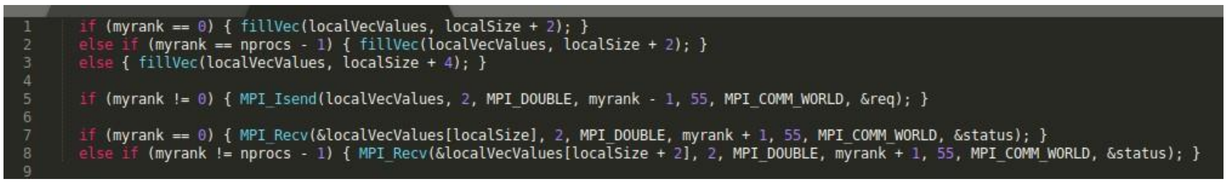
```
if (myrank == 0) { fillVec(VecValues, size); }

if (myrank == 0) { MPI_Isend(VecValues, localSize + 2, MPI_DOUBLE, myrank, myrank, MPI_COMM_WORLD, &req);
  for (i = 1; i < nprocs - 1; ++i) {
    MPI_Isend(&VecValues[localSize * i - 2], localSize + 4, MPI_DOUBLE, i, i, MPI_COMM_WORLD, &req);
  }
  MPI_Isend(&VecValues[localSize * (nprocs - 1) - 2], localSize + 2, MPI_DOUBLE, nprocs - 1, nprocs - 1, MPI_COMM_WORLD, &req);
}

if (myrank == 0) { MPI_Recv(localVecValues, localSize + 2, MPI_DOUBLE, 0, myrank, MPI_COMM_WORLD, &status); }
else if (myrank == (nprocs - 1)) { MPI_Recv(localVecValues, localSize + 2, MPI_DOUBLE, 0, myrank, MPI_COMM_WORLD, &status); }
else { MPI_Recv(localVecValues, localSize + 4, MPI_DOUBLE, 0, myrank, MPI_COMM_WORLD, &status); }
```

Figure 9 : Filling vector on master and distribute the others

The second algorithm can be seen at Figure 10.



```
if (myrank == 0) { fillVec(localVecValues, localSize + 2); }
else if (myrank == nprocs - 1) { fillVec(localVecValues, localSize + 2); }
else { fillVec(localVecValues, localSize + 4); }

if (myrank != 0) { MPI_Isend(localVecValues, 2, MPI_DOUBLE, myrank - 1, 55, MPI_COMM_WORLD, &req); }

if (myrank == 0) { MPI_Recv(&localVecValues[localSize], 2, MPI_DOUBLE, myrank + 1, 55, MPI_COMM_WORLD, &status); }
else if (myrank != nprocs - 1) { MPI_Recv(&localVecValues[localSize + 2], 2, MPI_DOUBLE, myrank + 1, 55, MPI_COMM_WORLD, &status); }
```

Figure 10 : Filling vector on each processor and share their same

When Amdahl’s formula is applied onto them, Figure 11 was produced. This graph shows us algorithm change seriously effect the speedup.

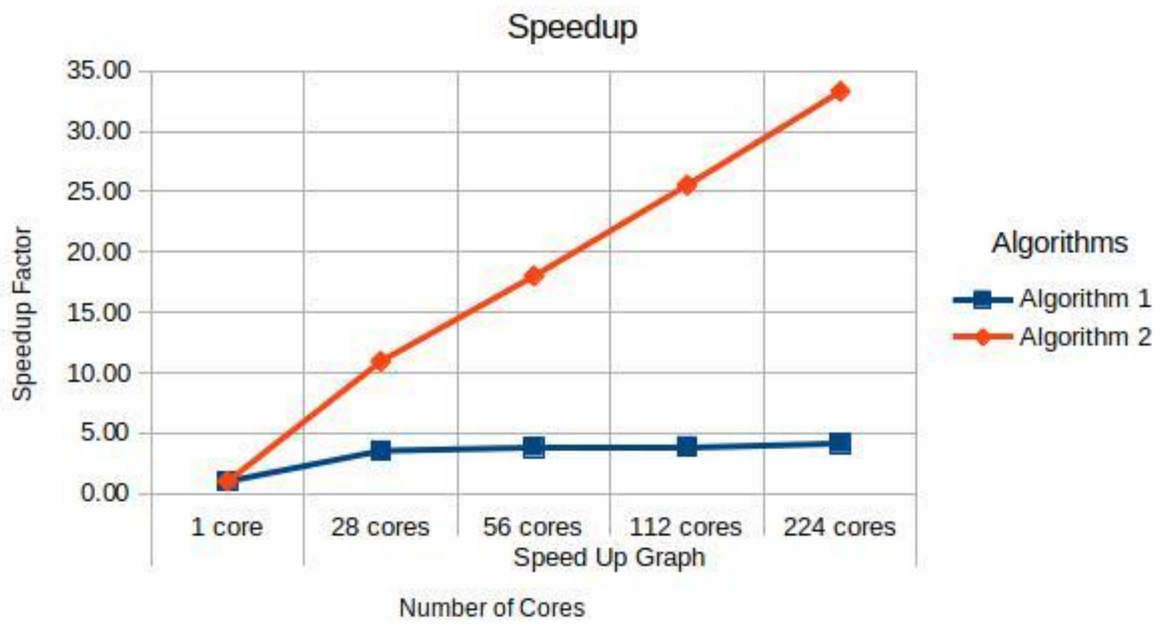


Figure 11 : Algorithm comparison

3.2.2 Gustafson’s Law and Effective Speedup

Theoretical speedup in latency of the execution can be calculated with Gustafson’s Law. Gustafson’s Law can be formulated like below:

$P_s = N + (1 - N) \times s$ **(Equation 3)**

where;

P_s = Performance speedup

s = Serial fraction

N = Number of cores

When the formula applied with the values we got from Amdahl’s Law, a graph comes up like Figure 12.

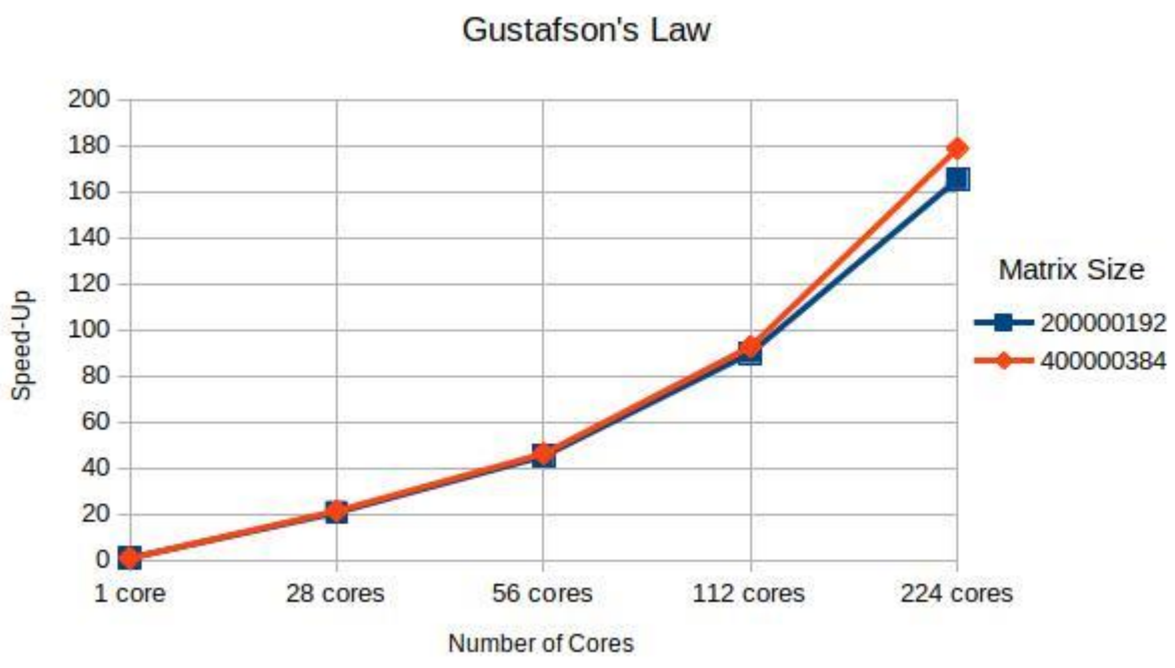


Figure 12 : Gustafson’s Law Graph

3.3. Floating Point Operations

Floating point operation per second is a way of sowing us the performance of the code.

For example, 1001280 x 1001280 matrix there are 1001280 x 5 – 6 = 5006394 nonzero values. These nonzeros multiplied with vector values and added into previous result value. Then 5006394 x 2 = 10012788 floating point operations done. Multiplication lasts 0.0010 seconds on 28 cores. So floating point operation per second can be calculated as 10012788 / 0.0010 = 10012788000 flops/sec = 10 Gflop/sec. If flops/sec calculated for all data, Figure 13 shows up.

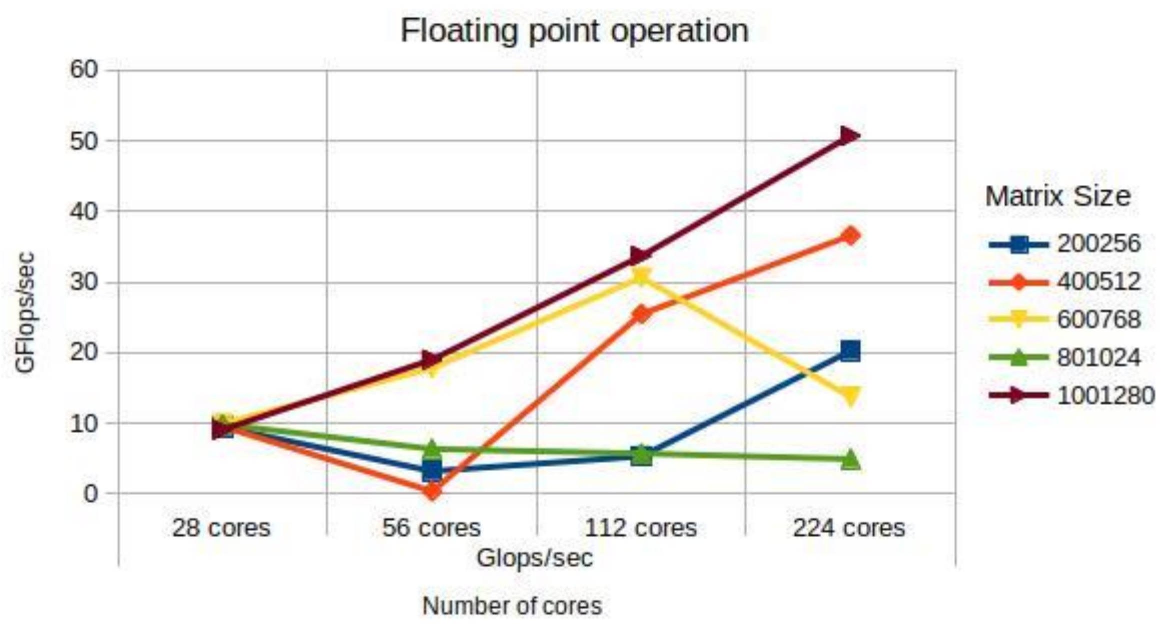


Figure 13 : Floating point operations per second

This graph helps us to understand which of the points is the peak point. For example peak point for 1001280 matrix size is the last point but, peak point for 600768 matrix size is the third point.

3.4. Hardware Metrics and Effects

Hardware	Value	Effect
Intel® Xeon® Processor E5-2680 v4		
L1 Cache instruction	32K	This effects the multiplication time very seriously because array sizes so big and program can not cached array effectively. That caused cache miss.
L1 Cache data	32 K	
L2 Cache	256 K	
L3 Cache	35840 K	
Number of Core	28	Inner machine communication is faster than inter-machine communication. This effects speed up factor while scaling problem into higher number of cores.
Bus Speed	9.6 GT/s	Gathering part got effected from bus speed.
Frequency	2.4 – 3.3 GHz	
Infiniband FDR Network		
Rate	56 Gb/s	This is important for inter-machine communications.
Memory		
Size (per machine)	128 GB	Memory latency effects the communication between cores. Cores had to communicate each other because of symmetry.

Scalability and efficiency is limited because cores need to communicate each other. On the other way array sizes are huge and that’s because whether communication or serial fraction increases.

4. RESULTS AND DISCUSSION

These graphs and results shows us problem did not parallelize well. First of all, for small sizes parallelizing this problem is not effective because MPI initilization part lasts way more longer than other parts.

Secondly, distribution could be better. Block distribution used in this project but cyclic distribution would be better and hybrid distribution would be the best. But it would increase the communication and scheduling.

Thirdly, serial fractions in the parallel section effects the speedup very badly. This problem could be overcome by better distribution and different algorithms.

REFERENCES

1. Buluç, Aydın; Fineman, Jeremy T.; Frigo, Matteo; Gilbert, John R.; Leiserson, Charles E. (2009). *Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks.*