Indian Institute of Information Technology Allahabad

PDC COURSE PROJECT

Sparse Matrix Dense Vector Multiplication (SpMV)

Supervisor - Dr. Anshu S Anand

Submitted by :

IIT2019019 Biswajeet Das
IIT2019021 Medha Balani
IIT2019015 Akash Anand
IIT2019193 Chetan Patidar

Table of Content

Implementation Overview:

- Multicore processing using **pThread**
- Multicore processing using **openMP**
- GPU processing using **CUDA**
- Distributed computing using **MPI**
- Distributed and parallel processing using **MPI+pThread** and **MPI+openMP**

Summary of contributions:

Akash Anand:
Sequential Algorithm

Chetan Patidar:
Pthread Parallel Algorithm 1 (using pthread and openMP)

Medha Balani:
Parallel Algorithm 2 (using pthread and openMP)
Parallel Algorithm 3 (using pthread, openMP and CUDA)

Biswajeet Das:
Parallel Algorithm 4 (using pthread and openMP along with Hillis Algorithm)
Parallel Algorithm 3 (using MPI, MPI+pthread and MPI+openMP)

Introduction:

SpMV is multiplication between a sparse matrix and a dense vector resulting in a dense vector of the form y = Ax. Sparse matrix is a matrix where most of its elements are zero.

As most of the elements in sparse matrix are zero, there is wastage of computation so A can be preprocessed to reduce both sequential and parallel runtime.

Applications:

- Used in many popular graph algorithms such as breadth first search, page rank, short-pathfinding and triangles counting.
- There is a large number of data sparsity in data samples and feature spaces, so SpMV is used in support-vector machines (SVM) and logistic regression in the Machine learning domain.
- It is a key computation in linear algebra.
- It is also used in solving large-scale sparse linear systems that arise from discretizations of partial differential equations.
- Used in many scientific computations such as graphics processing and conjugate gradients. [1]

Hence, it is crucial to study various algorithms to obtain high performance SpMV computations.

Algorithms:

The SpMV algorithms are affected by the representation of the sparse matrix. There are various formats such as CSR (Compressed Sparse Row), CSC (Compressed Sparse Column) formats, and Matrix Market Coordinate Format (MMCF).
The reason for using these formats is to remove redundant values i.e zero.
For our implementation, we are using CSR format to represent the sparse matrix.

CSR format: It contains three arrays- value array, column array and row offset array. Value array stores all non zero values in the sparse matrix in a row wise manner. The column array stores the column index of all non zero elements. Row offset array stores the number of non zero values till a particular row index. [2]

[1] Sequential

1. Convert sparse matrix into CSR format.
2. For every row, calculate the number of non zero elements in that particular row.
3. Loop through the non zero elements for every row.
4. Multiply the non zero value with the corresponding value in the dense vector.
5. Add the result to the row of the resultant vector.

```
for i = 0 to N-1 do
        for k = rowOffset[i] to k < rowOffset[i+1] do
                y[i] += value[k] * vector[column[k]]
        end for
end for
```

Execution time of the program: 0.001508 second.

```
The sequential answer is:
0.755918 -0.325148 0.00103017 1.26242 0.252252 -1.49809 -6.79878 2.39239 3.16555 -1.49186 -0.657184 -1.60807 0.3783
94 -0.748113 -0.0154968 0.457518 0.670194 -13.7464 0.474922 0.5395 -7.60159 -0.0699254 0.0210106 1.34126 0.243773 -
2.93591 5.1992 -2.84595 1.83699 2.22816 1.81867 1.31801 -0.519863 0.223857 -0.567833 0.0165912 2.02677 1.50911 13.8
692 -2.72047 -1.66686 6.42378 -2.98748 5.41084 26.6226 -4.4864 -2.00862 1.35354 -4.83743 4.9911 45.3386 1.8012 4.74
796 -35.7178 -3.82431 5.9771 -23.867 2.42517 1.39467 8.21859 0.432578 -0.00318072 31.3845 -1.11912 0.481357 9.74601
 -0.681514 -0.232873 0.0213747 0.736132 0.0272722 -1.5241 0.983109 0.22865 -5.67805 -0.0469082 0.350175 3.76285 0.0
423265 0.146179 5.25021 -0.859184 -0.827984 -2.48407 0.268577 -0.541081 1.60469 -0.510899 -0.368184 -1.91057 -3.866
54 0.679605 -6.55086 2.30441 -2.27205 -9.45656 -4.68618 3.39554 23.2404 1.94275 -0.985837 28.1608 1.39576 -4.23256
30.7589 1.77477 -4.43803 35.8965 1.29992 3.98559 -13.6084 -5.04825 3.39716 31.081 0.17257 0.303916 -0.251996 1.0009
9 0.606965 1.83105 -0.17105 0.735471 1.31551 -0.73438 0.803324 1.89861 -0.874013 0.937502 1.09508 0.618092 -0.35540
9 -5.73256 0.710163 0.959974 0.932712 0.30667 -0.0495628 5.35799
```

[2] Parallel Algorithm 1 using pthread.

1. Convert sparse matrix into CSR format.
2. Divide row operations into threads using pthread to perform the execution in parallel. Each thread performs rows / MAX_THREADS operations in parallel.
   If MAX_THREADS = 4
   Thread 1 operates on row 0, 4, 8 and so on.
   Thread 2 operates on row 1, 5, 9 and so on.
   Thread 3 operates on row 2, 6, 10 and so on.
   Thread 4 operates on row 3, 7, 11 and so on.
3. For every row, calculate the number of non zero elements in that particular row.
4. Loop through the non zero elements for every row.
5. Multiply the non zero value with the corresponding value in the dense vector.
6. Add the result to the row of the resultant vector.

Execution time using 1 thread: 0.000214 second.
Execution time using 2 threads: 0.000036 second.
Execution time using 4 threads: 0.000027 second.
Execution time using 8 threads: 0.000271 second.
Execution time using 16 threads: 0.001342 second.

```
Sequential answer:
0.755918 -0.325148 0.00103017 1.26242 0.252252 -1.49809 -6.79878 2.39239 3.16555 -1.49186 -0.657184 -1.60807 0.378394 -0.7
48113 -0.0154968 0.457518 0.670194 -13.7464 0.474922 0.5395 -7.60159 -0.0699254 0.0210106 1.34126 0.243773 -2.93591 5.1992
 -2.84595 1.83699 2.22816 1.81867 1.31801 -0.519863 0.223857 -0.567833 0.0165912 2.02677 1.50911 13.8692 -2.72047 -1.66686
 6.42378 -2.98748 5.41084 26.6226 -4.4864 -2.00862 1.35354 -4.83743 4.9911 45.3386 1.8012 4.74796 -35.7178 -3.82431 5.9771
 -23.867 2.42517 1.39467 8.21859 0.432578 -0.00318072 31.3845 -1.11912 0.481357 9.74601 -0.681514 -0.232873 0.0213747 0.73
6132 0.0272722 -1.5241 0.983109 0.22865 -5.67805 -0.0469082 0.350175 3.76285 0.0423265 0.146179 5.25021 -0.859184 -0.82798
4 -2.48407 0.268577 -0.541081 1.60469 -0.510899 -0.368184 -1.91057 -3.86654 0.679605 -6.55086 2.30441 -2.27205 -9.45656 -4
.68618 3.39554 23.2404 1.94275 -0.985837 28.1608 1.39576 -4.23256 30.7589 1.77477 -4.43803 35.8965 1.29992 3.98559 -13.608
4 -5.04825 3.39716 31.081 0.17257 0.303916 -0.251996 1.00099 0.606965 1.83105 -0.17105 0.735471 1.31551 -0.73438 0.803324
1.89861 -0.874013 0.937502 1.09508 0.618092 -0.355409 -5.73256 0.710163 0.959974 0.932712 0.30667 -0.0495628 5.35799

Final answer:
0.755918 -0.325148 0.00103017 1.26242 0.252252 -1.49809 -6.79878 2.39239 3.16555 -1.49186 -0.657184 -1.60807 0.378394 -0.7
48113 -0.0154968 0.457518 0.670194 -13.7464 0.474922 0.5395 -7.60159 -0.0699254 0.0210106 1.34126 0.243773 -2.93591 5.1992
 -2.84595 1.83699 2.22816 1.81867 1.31801 -0.519863 0.223857 -0.567833 0.0165912 2.02677 1.50911 13.8692 -2.72047 -1.66686
 6.42378 -2.98748 5.41084 26.6226 -4.4864 -2.00862 1.35354 -4.83743 4.9911 45.3386 1.8012 4.74796 -35.7178 -3.82431 5.9771
 -23.867 2.42517 1.39467 8.21859 0.432578 -0.00318072 31.3845 -1.11912 0.481357 9.74601 -0.681514 -0.232873 0.0213747 0.73
6132 0.0272722 -1.5241 0.983109 0.22865 -5.67805 -0.0469082 0.350175 3.76285 0.0423265 0.146179 5.25021 -0.859184 -0.82798
4 -2.48407 0.268577 -0.541081 1.60469 -0.510899 -0.368184 -1.91057 -3.86654 0.679605 -6.55086 2.30441 -2.27205 -9.45656 -4
.68618 3.39554 23.2404 1.94275 -0.985837 28.1608 1.39576 -4.23256 30.7589 1.77477 -4.43803 35.8965 1.29992 3.98559 -13.608
4 -5.04825 3.39716 31.081 0.17257 0.303916 -0.251996 1.00099 0.606965 1.83105 -0.17105 0.735471 1.31551 -0.73438 0.803324
1.89861 -0.874013 0.937502 1.09508 0.618092 -0.355409 -5.73256 0.710163 0.959974 0.932712 0.30667 -0.0495628 5.35799

Correct

Time Taken using 16 threads: 0.001342 sec
```

[3] Parallel Algorithm 1 using openmp.

1. Convert sparse matrix into CSR format.
2. Divide row operations into threads using openMP syntax (#pragma omp parallel num_threads(MAX_THREADS)) to perform the execution in parallel.
3. For every row, calculate the number of non zero elements in that particular row.
4. Loop through the non zero elements for every row.
5. Multiply the non zero value with the corresponding value in the dense vector.
6. Add the result to the row of the resultant vector.

Execution time using 1 thread: 0.000206 second.
Execution time using 2 threads: 0.000034 second.
Execution time using 4 threads: 0.000023 second.
Execution time using 8 threads: 0.000269 second.
Execution time using 16 threads: 0.001326 second.

```
Sequential answer:
0.755918 -0.325148 0.00103017 1.26242 0.252252 -1.49809 -6.79878 2.39239 3.16555 -1.49186 -0.657184 -1.60807 0.378394 -0.7
48113 -0.0154968 0.457518 0.670194 -13.7464 0.474922 0.5395 -7.60159 -0.0699254 0.0210106 1.34126 0.243773 -2.93591 5.1992
 -2.84595 1.83699 2.22816 1.81867 1.31801 -0.519863 0.223857 -0.567833 0.0165912 2.02677 1.50911 13.8692 -2.72047 -1.66686
 6.42378 -2.98748 5.41084 26.6226 -4.4864 -2.00862 1.35354 -4.83743 4.9911 45.3386 1.8012 4.74796 -35.7178 -3.82431 5.9771
 -23.867 2.42517 1.39467 8.21859 0.432578 -0.00318072 31.3845 -1.11912 0.481357 9.74601 -0.681514 -0.232873 0.0213747 0.73
6132 0.0272722 -1.5241 0.983109 0.22865 -5.67805 -0.0469082 0.350175 3.76285 0.0423265 0.146179 5.25021 -0.859184 -0.82798
4 -2.48407 0.268577 -0.541081 1.60469 -0.510899 -0.368184 -1.91057 -3.86654 0.679605 -6.55086 2.30441 -2.27205 -9.45656 -4
.68618 3.39554 23.2404 1.94275 -0.985837 28.1608 1.39576 -4.23256 30.7589 1.77477 -4.43803 35.8965 1.29992 3.98559 -13.608
4 -5.04825 3.39716 31.081 0.17257 0.303916 -0.251996 1.00099 0.606965 1.83105 -0.17105 0.735471 1.31551 -0.73438 0.803324
1.89861 -0.874013 0.937502 1.09508 0.618092 -0.355409 -5.73256 0.710163 0.959974 0.932712 0.30667 -0.0495628 5.35799

Final answer:
0.755918 -0.325148 0.00103017 1.26242 0.252252 -1.49809 -6.79878 2.39239 3.16555 -1.49186 -0.657184 -1.60807 0.378394 -0.7
48113 -0.0154968 0.457518 0.670194 -13.7464 0.474922 0.5395 -7.60159 -0.0699254 0.0210106 1.34126 0.243773 -2.93591 5.1992
 -2.84595 1.83699 2.22816 1.81867 1.31801 -0.519863 0.223857 -0.567833 0.0165912 2.02677 1.50911 13.8692 -2.72047 -1.66686
 6.42378 -2.98748 5.41084 26.6226 -4.4864 -2.00862 1.35354 -4.83743 4.9911 45.3386 1.8012 4.74796 -35.7178 -3.82431 5.9771
 -23.867 2.42517 1.39467 8.21859 0.432578 -0.00318072 31.3845 -1.11912 0.481357 9.74601 -0.681514 -0.232873 0.0213747 0.73
6132 0.0272722 -1.5241 0.983109 0.22865 -5.67805 -0.0469082 0.350175 3.76285 0.0423265 0.146179 5.25021 -0.859184 -0.82798
4 -2.48407 0.268577 -0.541081 1.60469 -0.510899 -0.368184 -1.91057 -3.86654 0.679605 -6.55086 2.30441 -2.27205 -9.45656 -4
.68618 3.39554 23.2404 1.94275 -0.985837 28.1608 1.39576 -4.23256 30.7589 1.77477 -4.43803 35.8965 1.29992 3.98559 -13.608
4 -5.04825 3.39716 31.081 0.17257 0.303916 -0.251996 1.00099 0.606965 1.83105 -0.17105 0.735471 1.31551 -0.73438 0.803324
1.89861 -0.874013 0.937502 1.09508 0.618092 -0.355409 -5.73256 0.710163 0.959974 0.932712 0.30667 -0.0495628 5.35799

Correct

Time Taken using 16 threads: 0.001326 sec
```

[4] Parallel Algorithm 2 using pthread.

1. Convert sparse matrix into CSR format.
2. Divide row operations into threads to perform the execution in parallel using pthread. Each thread operates on a continuous segment of length rows / MAX_THREADS.
   If MAX_THREADS = 4 and rows = 16
   Thread 1 operates on row 0, 1, 2 and so on.
   Thread 2 operates on row 4, 5, 6 and so on.
   Thread 3 operates on row 8, 9, 10 and so on.

Thread 4 operates on row 12, 13, 14 and so on.

3. For every row, calculate the number of non zero elements in that particular row.
4. Loop through the non zero elements for every row.
5. Multiply the non zero value with the corresponding value in the dense vector.
6. Add the result to the row of the resultant vector.

Execution time using 1 thread: 0.000299 second.
Execution time using 2 threads: 0.000285 second.
Execution time using 4 threads: 0.000124 second.
Execution time using 8 threads: 0.000102 second.
Execution time using 16 threads: 0.000132 second.

```
Sequential answer:
0.755918 -0.325148 0.00103017 1.26242 0.252252 -1.49809 -6.79878 2.39239 3.16555 -1.49186 -0.657184 -1.60807 0.378394 -0.7
48113 -0.0154968 0.457518 0.670194 -13.7464 0.474922 0.5395 -7.60159 -0.0699254 0.0210106 1.34126 0.243773 -2.93591 5.1992
 -2.84595 1.83699 2.22816 1.81867 1.31801 -0.519863 0.223857 -0.567833 0.0165912 2.02677 1.50911 13.8692 -2.72047 -1.66686
 6.42378 -2.98748 5.41084 26.6226 -4.4864 -2.00862 1.35354 -4.83743 4.9911 45.3386 1.8012 4.74796 -35.7178 -3.82431 5.9771
 -23.867 2.42517 1.39467 8.21859 0.432578 -0.00318072 31.3845 -1.11912 0.481357 9.74601 -0.681514 -0.232873 0.0213747 0.73
6132 0.0272722 -1.5241 0.983109 0.22865 -5.67805 -0.0469082 0.350175 3.76285 0.0423265 0.146179 5.25021 -0.859184 -0.82798
4 -2.48407 0.268577 -0.541081 1.60469 -0.510899 -0.368184 -1.91057 -3.86654 0.679605 -6.55086 2.30441 -2.27205 -9.45656 -4
.68618 3.39554 23.2404 1.94275 -0.985837 28.1608 1.39576 -4.23256 30.7589 1.77477 -4.43803 35.8965 1.29992 3.98559 -13.608
4 -5.04825 3.39716 31.081 0.17257 0.303916 -0.251996 1.00099 0.606965 1.83105 -0.17105 0.735471 1.31551 -0.73438 0.803324
1.89861 -0.874013 0.937502 1.09508 0.618092 -0.355409 -5.73256 0.710163 0.959974 0.932712 0.30667 -0.0495628 5.35799

Final answer:
0.755918 -0.325148 0.00103017 1.26242 0.252252 -1.49809 -6.79878 2.39239 3.16555 -1.49186 -0.657184 -1.60807 0.378394 -0.7
48113 -0.0154968 0.457518 0.670194 -13.7464 0.474922 0.5395 -7.60159 -0.0699254 0.0210106 1.34126 0.243773 -2.93591 5.1992
 -2.84595 1.83699 2.22816 1.81867 1.31801 -0.519863 0.223857 -0.567833 0.0165912 2.02677 1.50911 13.8692 -2.72047 -1.66686
 6.42378 -2.98748 5.41084 26.6226 -4.4864 -2.00862 1.35354 -4.83743 4.9911 45.3386 1.8012 4.74796 -35.7178 -3.82431 5.9771
 -23.867 2.42517 1.39467 8.21859 0.432578 -0.00318072 31.3845 -1.11912 0.481357 9.74601 -0.681514 -0.232873 0.0213747 0.73
6132 0.0272722 -1.5241 0.983109 0.22865 -5.67805 -0.0469082 0.350175 3.76285 0.0423265 0.146179 5.25021 -0.859184 -0.82798
4 -2.48407 0.268577 -0.541081 1.60469 -0.510899 -0.368184 -1.91057 -3.86654 0.679605 -6.55086 2.30441 -2.27205 -9.45656 -4
.68618 3.39554 23.2404 1.94275 -0.985837 28.1608 1.39576 -4.23256 30.7589 1.77477 -4.43803 35.8965 1.29992 3.98559 -13.608
4 -5.04825 3.39716 31.081 0.17257 0.303916 -0.251996 1.00099 0.606965 1.83105 -0.17105 0.735471 1.31551 -0.73438 0.803324
1.89861 -0.874013 0.937502 1.09508 0.618092 -0.355409 -5.73256 0.710163 0.959974 0.932712 0.30667 -0.0495628 5.35799

Correct

Time Taken using 2 threads: 0.000285 sec
```

[5] Parallel Algorithm 2 using openMP.

1. Convert sparse matrix into CSR format.
2. Divide row operations into threads using openMP syntax (#pragma omp parallel num_threads(MAX_THREADS) for) to perform the execution in parallel.
3. For every row, calculate the number of non zero elements in that particular row.
4. Loop through the non zero elements for every row.
5. Multiply the non zero value with the corresponding value in the dense vector.
6. Add the result to the row of the resultant vector.

Execution time using 1 thread: 0.000032 second.
Execution time using 2 threads: 0.000021 second.
Execution time using 4 threads: 0.000014 second.

Execution time using 8 threads: 0.000009 second.
Execution time using 16 threads: 0.000007 second.

```
Sequential answer:
0.755918 -0.325148 0.00103017 1.26242 0.252252 -1.49809 -6.79878 2.39239 3.16555 -1.49186 -0.657184 -1.60807 0.378394 -0.
748113 -0.0154968 0.457518 0.670194 -13.7464 0.474922 0.5395 -7.60159 -0.0699254 0.0210106 1.34126 0.243773 -2.93591 5.19
92 -2.84595 1.83699 2.22816 1.81867 1.31801 -0.519863 0.223857 -0.567833 0.0165912 2.02677 1.50911 13.8692 -2.72047 -1.66
686 6.42378 -2.98748 5.41084 26.6226 -4.4864 -2.00862 1.35354 -4.83743 4.9911 45.3386 1.8012 4.74796 -35.7178 -3.82431 5.
9771 -23.867 2.42517 1.39467 8.21859 0.432578 -0.00318072 31.3845 -1.11912 0.481357 9.74601 -0.681514 -0.232873 0.0213747
 0.736132 0.0272722 -1.5241 0.983109 0.22865 -5.67805 -0.0469082 0.350175 3.76285 0.0423265 0.146179 5.25021 -0.859184 -0
.827984 -2.48407 0.268577 -0.541081 1.60469 -0.510899 -0.368184 -1.91057 -3.86654 0.679605 -6.55086 2.30441 -2.27205 -9.4
5656 -4.68618 3.39554 23.2404 1.94275 -0.985837 28.1608 1.39576 -4.23256 30.7589 1.77477 -4.43803 35.8965 1.29992 3.98559
 -13.6084 -5.04825 3.39716 31.081 0.17257 0.303916 -0.251996 1.00099 0.606965 1.83105 -0.17105 0.735471 1.31551 -0.73438
0.803324 1.89861 -0.874013 0.937502 1.09508 0.618092 -0.355409 -5.73256 0.710163 0.959974 0.932712 0.30667 -0.0495628 5.3
5799

Final answer:
0.755918 -0.325148 0.00103017 1.26242 0.252252 -1.49809 -6.79878 2.39239 3.16555 -1.49186 -0.657184 -1.60807 0.378394 -0.
748113 -0.0154968 0.457518 0.670194 -13.7464 0.474922 0.5395 -7.60159 -0.0699254 0.0210106 1.34126 0.243773 -2.93591 5.19
92 -2.84595 1.83699 2.22816 1.81867 1.31801 -0.519863 0.223857 -0.567833 0.0165912 2.02677 1.50911 13.8692 -2.72047 -1.66
686 6.42378 -2.98748 5.41084 26.6226 -4.4864 -2.00862 1.35354 -4.83743 4.9911 45.3386 1.8012 4.74796 -35.7178 -3.82431 5.
9771 -23.867 2.42517 1.39467 8.21859 0.432578 -0.00318072 31.3845 -1.11912 0.481357 9.74601 -0.681514 -0.232873 0.0213747
 0.736132 0.0272722 -1.5241 0.983109 0.22865 -5.67805 -0.0469082 0.350175 3.76285 0.0423265 0.146179 5.25021 -0.859184 -0
.827984 -2.48407 0.268577 -0.541081 1.60469 -0.510899 -0.368184 -1.91057 -3.86654 0.679605 -6.55086 2.30441 -2.27205 -9.4
5656 -4.68618 3.39554 23.2404 1.94275 -0.985837 28.1608 1.39576 -4.23256 30.7589 1.77477 -4.43803 35.8965 1.29992 3.98559
 -13.6084 -5.04825 3.39716 31.081 0.17257 0.303916 -0.251996 1.00099 0.606965 1.83105 -0.17105 0.735471 1.31551 -0.73438
0.803324 1.89861 -0.874013 0.937502 1.09508 0.618092 -0.355409 -5.73256 0.710163 0.959974 0.932712 0.30667 -0.0495628 5.3
5799

Correct

Time Taken using 8 threads: 0.000009 sec
```

[6] Parallel Algorithm 3 (binary search) using pthread.

1. Convert sparse matrix into CSR format.
2. Iterate through every non zero element by dividing operations between threads using pthread to execute in parallel where each thread works on (number_of_non_zero_elements/ MAX_THREADS) non-zero elements.
3. Find the row number from rowOffset corresponding to the non-zero values using binary search (lower_bound).
4. Multiply non zero value with dense vector index corresponding to the column index of the input matrix.
5. Add the result to the row index found in step 3 in the resultant vector.

Execution time using 1 thread: 0.000198 second.
Execution time using 2 threads: 0.000185 second.
Execution time using 4 threads: 0.000127 second.
Execution time using 8 threads: 0.000112 second.
Execution time using 16 threads: 0.000134 second.

```
The final answer is:
0.755918 -0.325148 0.00103017 1.26242 0.252252 -1.49809 -6.79878 2.39239 3.16555 -1.49186 -0.657184 -1.60807 0.378394 -0.
748113 -0.0154968 0.457518 0.670194 -13.7464 0.474922 0.5395 -7.60159 -0.0699254 0.0210106 1.34126 0.243773 -2.93591 5.19
92 -2.84595 1.83699 2.22816 1.81867 1.31801 -0.519863 0.223857 -0.567833 0.0165912 2.02677 1.50911 13.8692 -2.72047 -1.66
686 6.42378 -2.98748 5.41084 26.6226 -4.4864 -2.00862 1.35354 -4.83743 4.9911 45.3386 1.8012 4.74796 -35.7178 -3.82431 5.
9771 -23.867 2.42517 1.39467 8.21859 0.432578 -0.00318072 31.3845 -1.11912 0.481357 9.74601 -0.681514 -0.232873 0.0213747
 0.736132 0.0272722 -1.5241 0.983109 0.22865 -5.67805 -0.0469082 0.350175 3.76285 0.0423265 0.146179 5.25021 -0.859184 -0
.827984 -2.48407 0.268577 -0.541081 1.60469 -0.510899 -0.368184 -1.91057 -3.86654 0.679605 -6.55086 2.30441 -2.27205 -9.4
5656 -4.68618 3.39554 23.2404 1.94275 -0.985837 28.1608 1.39576 -4.23256 30.7589 1.77477 -4.43803 35.8965 1.29992 3.98559
 -13.6084 -5.04825 3.39716 31.081 0.17257 0.303916 -0.251996 1.00099 0.606965 1.83105 -0.17105 0.735471 1.31551 -0.73438
0.803324 1.89861 -0.874013 0.937502 1.09508 0.618092 -0.355409 -5.73256 0.710163 0.959974 0.932712 0.30667 -0.0495628 5.3
5799

The sequential answer is:
0.755918 -0.325148 0.00103017 1.26242 0.252252 -1.49809 -6.79878 2.39239 3.16555 -1.49186 -0.657184 -1.60807 0.378394 -0.
748113 -0.0154968 0.457518 0.670194 -13.7464 0.474922 0.5395 -7.60159 -0.0699254 0.0210106 1.34126 0.243773 -2.93591 5.19
92 -2.84595 1.83699 2.22816 1.81867 1.31801 -0.519863 0.223857 -0.567833 0.0165912 2.02677 1.50911 13.8692 -2.72047 -1.66
686 6.42378 -2.98748 5.41084 26.6226 -4.4864 -2.00862 1.35354 -4.83743 4.9911 45.3386 1.8012 4.74796 -35.7178 -3.82431 5.
9771 -23.867 2.42517 1.39467 8.21859 0.432578 -0.00318072 31.3845 -1.11912 0.481357 9.74601 -0.681514 -0.232873 0.0213747
 0.736132 0.0272722 -1.5241 0.983109 0.22865 -5.67805 -0.0469082 0.350175 3.76285 0.0423265 0.146179 5.25021 -0.859184 -0
.827984 -2.48407 0.268577 -0.541081 1.60469 -0.510899 -0.368184 -1.91057 -3.86654 0.679605 -6.55086 2.30441 -2.27205 -9.4
5656 -4.68618 3.39554 23.2404 1.94275 -0.985837 28.1608 1.39576 -4.23256 30.7589 1.77477 -4.43803 35.8965 1.29992 3.98559
 -13.6084 -5.04825 3.39716 31.081 0.17257 0.303916 -0.251996 1.00099 0.606965 1.83105 -0.17105 0.735471 1.31551 -0.73438
0.803324 1.89861 -0.874013 0.937502 1.09508 0.618092 -0.355409 -5.73256 0.710163 0.959974 0.932712 0.30667 -0.0495628 5.3
5799

Correct
```

[7] Parallel Algorithm 3 using openMP.

1. Convert sparse matrix into CSR format.
2. Iterate through every non zero element by dividing operations between threads using openMP (#pragma omp parallel num_threads(MAX_THREADS)) to execute in parallel.
3. Find the row number from rowOffset corresponding to the non-zero values using binary search (lower_bound).
4. Multiply non zero value with dense vector index corresponding to the column index of the input matrix.
5. Add the result to the row index found in step 3 in the resultant vector.

Execution time using 1 thread: 0.000231 second.
Execution time using 2 threads: 0.000215 second.
Execution time using 4 threads: 0.000195 second.
Execution time using 8 threads: 0.000183 second.
Execution time using 16 threads: 0.000231 second.

```
The final answer is:
0.755918 -0.325148 0.00103017 1.26242 0.252252 -1.49809 -6.79878 2.39239 3.16555 -1.49186 -0.657184 -1.60807 0.378394 -
0.748113 -0.0154968 0.457518 0.670194 -13.7464 0.474922 0.5395 -7.60159 -0.0699254 0.0210106 1.34126 0.243773 -2.93591
5.1992 -2.84595 1.83699 2.22816 1.81867 1.31801 -0.519863 0.223857 -0.567833 0.0165912 2.02677 1.50911 13.8692 -2.72047
 -1.66686 6.42378 -2.98748 5.41084 26.6226 -4.4864 -2.00862 1.35354 -4.83743 4.9911 45.3386 1.8012 4.74796 -35.7178 -3.
82431 5.9771 -23.867 2.42517 1.39467 8.21859 0.432578 -0.00318072 31.3845 -1.11912 0.481357 9.74601 -0.681514 -0.232873
 0.0213747 0.736132 0.0272722 -1.5241 0.983109 0.22865 -5.67805 -0.0469082 0.350175 3.76285 0.0423265 0.146179 5.25021
-0.859184 -0.827984 -2.48407 0.268577 -0.541081 1.60469 -0.510899 -0.368184 -1.91057 -3.86654 0.679605 -6.55086 2.30441
 -2.27205 -9.45656 -4.68618 3.39554 23.2404 1.94275 -0.985837 28.1608 1.39576 -4.23256 30.7589 1.77477 -4.43803 35.8965
 1.29992 3.98559 -13.6084 -5.04825 3.39716 31.081 0.17257 0.303916 -0.251996 1.00099 0.606965 1.83105 -0.17105 0.735471
 1.31551 -0.73438 0.803324 1.89861 -0.874013 0.937502 1.09508 0.618092 -0.355409 -5.73256 0.710163 0.959974 0.932712 0.
30667 -0.0495628 5.35799

The sequential answer is:
0.755918 -0.325148 0.00103017 1.26242 0.252252 -1.49809 -6.79878 2.39239 3.16555 -1.49186 -0.657184 -1.60807 0.378394 -
0.748113 -0.0154968 0.457518 0.670194 -13.7464 0.474922 0.5395 -7.60159 -0.0699254 0.0210106 1.34126 0.243773 -2.93591
5.1992 -2.84595 1.83699 2.22816 1.81867 1.31801 -0.519863 0.223857 -0.567833 0.0165912 2.02677 1.50911 13.8692 -2.72047
 -1.66686 6.42378 -2.98748 5.41084 26.6226 -4.4864 -2.00862 1.35354 -4.83743 4.9911 45.3386 1.8012 4.74796 -35.7178 -3.
82431 5.9771 -23.867 2.42517 1.39467 8.21859 0.432578 -0.00318072 31.3845 -1.11912 0.481357 9.74601 -0.681514 -0.232873
 0.0213747 0.736132 0.0272722 -1.5241 0.983109 0.22865 -5.67805 -0.0469082 0.350175 3.76285 0.0423265 0.146179 5.25021
-0.859184 -0.827984 -2.48407 0.268577 -0.541081 1.60469 -0.510899 -0.368184 -1.91057 -3.86654 0.679605 -6.55086 2.30441
 -2.27205 -9.45656 -4.68618 3.39554 23.2404 1.94275 -0.985837 28.1608 1.39576 -4.23256 30.7589 1.77477 -4.43803 35.8965
 1.29992 3.98559 -13.6084 -5.04825 3.39716 31.081 0.17257 0.303916 -0.251996 1.00099 0.606965 1.83105 -0.17105 0.735471
 1.31551 -0.73438 0.803324 1.89861 -0.874013 0.937502 1.09508 0.618092 -0.355409 -5.73256 0.710163 0.959974 0.932712 0.
30667 -0.0495628 5.35799

Correct
Time taken by program using 4 threads is 0.000195 sec
```

[8] Parallel Algorithm 3 using CUDA.

1. Input sparse matrix from the file in CSR format
2. Allocate size of all the host variables and initialize them using the N, M input
3. Create row offset array using prefix sum
4. Input dense vector from file
5. Allocate memory in GPU using cudaMalloc for all the required variables.
6. Copy values from host to device for all the arrays for further calculation
7. Call the kernel and pass values to calculate answer in GPU
8. Copy back the answer from device to host
9. Print the answer

Time taken by program using 1 blocks and 1 threads is 0.379053 sec
Time taken by program using 2 blocks and 128 threads is 0.223105 sec
Time taken by program using 4 blocks and 2 threads is 0.277948 sec
Time taken by program using 4 blocks and 64 threads is 0.276365 sec
Time taken by program using 8 blocks and 32 threads is 0.277838 sec
Time taken by program using 16 blocks and 16 threads is 0.210272 sec

```
Sequential Answer:
0.755918 -0.325148 0.00103017 1.26242 0.252252 -1.49809 -6.79878 2.39239 3.16555 -1.49186 -0.657184

Final Answer using CUDA:
0.755918 -0.325148 0.00103017 1.26242 0.252252 -1.49809 -6.79878 2.39239 3.16555 -1.49186 -0.657184

Time taken using 8 blocks and 2 threads is 0.129727 sec
```

[9] Parallel Algorithm 3 using MPI

I. Create an MPI environment.
II. Master Process:
    1. Input sparse matrix in CSR format.
    2. Create row offset array using prefix sum
    3. Input dense vector from file
    4. Divide the answer calculation into equal blocks and send required data(used to calculate answer) to child processes.
    5. Data passed: rowOffset, value(sub-array), b, startIndex, column(sub-array)
    6. Calculate partial answer for it's own sub-problem
    7. Receive the partial answer from the child processes.
    8. Add the partial answer to the final answer.
III. Child Processes
    1. Receive the data from the master process.
    2. Apply SPMV algorithm using binary search to calculate partial answer.
    3. Finally send the partial answer to the master process
IV. Clear MPI environment.

Execution time using 4 processes: 0.000476 seconds
Execution time using 2 processes: 0.000442 seconds
Execution time using 1 process: 0.000470 seconds

```
The final answer is:
0.755918 -0.325148 0.001030 1.262419 0.252252 -1.498094 -6.798784 2.392391 3.165552 -1.491856 -0.657184 -1.608070 0.378394 -0.748113 -0.0
15497 0.457518 0.670194 -13.746393 0.474922 0.539500 -7.601586 -0.069925 0.021011 1.341256 0.243773 -2.935911 5.199205 -2.845950 1.836988
 2.228165 1.818666 1.318007 -0.519863 0.223857 -0.567833 0.016591 2.026767 1.509110 13.869150 -2.720471 -1.666859 6.423777 -2.987484 5.41
0840 26.622616 -4.486403 -2.008623 1.353536 -4.837431 4.991103 45.338573 1.801199 4.747960 -35.717758 -3.824305 5.977098 -23.867020 2.425
172 1.394668 8.218590 0.432578 -0.003181 31.384497 -1.119125 0.481357 9.746015 -0.681514 -0.232873 0.021375 0.736132 0.027272 -1.524101 0
.983109 0.228650 -5.678046 -0.046908 0.350175 3.762846 0.042326 0.146179 5.250207 -0.859184 -0.827984 -2.484071 0.268577 -0.541081 1.6046
91 -0.510899 -0.368184 -1.910566 -3.866541 0.679605 -6.550858 2.304412 -2.272054 -9.456560 -4.686183 3.395544 23.240416 1.942752 -0.98583
7 28.160833 1.395764 -4.232559 30.758883 1.774767 -4.438025 35.896500 1.299924 3.985591 -13.608424 -5.048252 3.397160 31.081030 0.172570
0.303916 -0.251996 1.000991 0.606965 1.831054 -0.171050 0.735471 1.315514 -0.734380 0.803324 1.898607 -0.874013 0.937502 1.095077 0.61809
2 -0.355409 -5.732562 0.710163 0.959974 0.932712 0.306670 -0.049563 5.357992

The sequential answer is:
0.755918 -0.325148 0.001030 1.262419 0.252252 -1.498094 -6.798784 2.392391 3.165552 -1.491856 -0.657184 -1.608070 0.378394 -0.748113 -0.0
15497 0.457518 0.670194 -13.746393 0.474922 0.539500 -7.601586 -0.069925 0.021011 1.341256 0.243773 -2.935911 5.199205 -2.845950 1.836988
 2.228165 1.818666 1.318007 -0.519863 0.223857 -0.567833 0.016591 2.026767 1.509110 13.869150 -2.720471 -1.666859 6.423777 -2.987484 5.41
0840 26.622616 -4.486403 -2.008623 1.353536 -4.837431 4.991103 45.338573 1.801199 4.747960 -35.717758 -3.824305 5.977098 -23.867020 2.425
172 1.394668 8.218590 0.432578 -0.003181 31.384497 -1.119125 0.481357 9.746015 -0.681514 -0.232873 0.021375 0.736132 0.027272 -1.524101 0
.983109 0.228650 -5.678046 -0.046908 0.350175 3.762846 0.042326 0.146179 5.250207 -0.859184 -0.827984 -2.484071 0.268577 -0.541081 1.6046
91 -0.510899 -0.368184 -1.910566 -3.866541 0.679605 -6.550858 2.304412 -2.272054 -9.456560 -4.686183 3.395544 23.240416 1.942752 -0.98583
7 28.160833 1.395764 -4.232559 30.758883 1.774767 -4.438025 35.896500 1.299924 3.985591 -13.608424 -5.048252 3.397160 31.081030 0.172570
0.303916 -0.251996 1.000991 0.606965 1.831054 -0.171050 0.735471 1.315514 -0.734380 0.803324 1.898607 -0.874013 0.937502 1.095077 0.61809
2 -0.355409 -5.732562 0.710163 0.959974 0.932712 0.306670 -0.049563 5.357992

***Correct***

Time taken: 0.000470 seconds
```

[10] Parallel Algorithm 3 using MPI + pthread

I. Create an MPI environment.
II. Master Process:
    1. Input sparse matrix in CSR format.
    2. Create row offset array using prefix sum
    3. Input dense vector from file
    4. Divide the answer calculation into equal blocks and send required data(used to calculate answer) to child processes.
    5. Data passed: rowOffset, value(sub-array), b, startIndex, column(sub-array)
    6. Calculate partial answer for it's own sub-problem
    7. Receive the partial answer from the child processes.
    8. Add the partial answer to the final answer.
III. Child Process:
    1. Receive the data from the master process.
    2. Apply SPMV algorithm using pthread library using binary search [used in pthread_algo3] on partial data to calculate partial answer.
    3. Finally send the partial answer to the master process
IV. Clear MPI environment.

Execution time using 4 processes and 1 thread: 0.000876 seconds
Execution time using 4 processes and 2 threads: 0.000394 seconds
Execution time using 4 processes and 4 threads: 0.000161 seconds
Execution time using 2 processes and 8 threads: 0.000219 seconds
Execution time using 2 processes and 16 threads: 0.000101 seconds

```
The final answer is:
0.755918 -0.325148 0.001030 1.262419 0.252252 -1.498094 -6.798784 2.392391 3.165552 -1.491856 -0.657184 -1.608070 0.378394 -0.748113 -0.0
15497 0.457518 0.670194 -13.746393 0.474922 0.539500 -7.601586 -0.069925 0.021011 1.341256 0.243773 -2.935911 5.199205 -2.845950 1.836988
 2.228165 1.818666 1.318007 -0.519863 0.223857 -0.567833 0.016591 2.026767 1.509110 13.869150 -2.720471 -1.666859 6.423777 -2.987484 5.41
0840 26.622616 -4.486403 -2.008623 1.353536 -4.837431 4.991103 45.338573 1.801199 4.747960 -35.717758 -3.824305 5.977098 -23.867020 2.425
172 1.394668 8.218590 0.432578 -0.003181 31.384497 -1.119125 0.481357 9.746015 -0.681514 -0.232873 0.021375 0.736132 0.027272 -1.524101 0
.983109 0.228650 -5.678046 -0.046908 0.350175 3.762846 0.042326 0.146179 5.250207 -0.859184 -0.827984 -2.484071 0.268577 -0.541081 1.6046
91 -0.510899 -0.368184 -1.910566 -3.866541 0.679605 -6.550858 2.304412 -2.272054 -9.456560 -4.686183 3.395544 23.240416 1.942752 -0.98583
7 28.160833 1.395764 -4.232559 30.758883 1.774767 -4.438025 35.896500 1.299924 3.985591 -13.608424 -5.048252 3.397160 31.081030 0.172570
0.303916 -0.251996 1.000991 0.606965 1.831054 -0.171050 0.735471 1.315514 -0.734380 0.803324 1.898607 -0.874013 0.937502 1.095077 0.61809
2 -0.355409 -5.732562 0.710163 0.959974 0.932712 0.306670 -0.049563 5.357992

The sequential answer is:
0.755918 -0.325148 0.001030 1.262419 0.252252 -1.498094 -6.798784 2.392391 3.165552 -1.491856 -0.657184 -1.608070 0.378394 -0.748113 -0.0
15497 0.457518 0.670194 -13.746393 0.474922 0.539500 -7.601586 -0.069925 0.021011 1.341256 0.243773 -2.935911 5.199205 -2.845950 1.836988
 2.228165 1.818666 1.318007 -0.519863 0.223857 -0.567833 0.016591 2.026767 1.509110 13.869150 -2.720471 -1.666859 6.423777 -2.987484 5.41
0840 26.622616 -4.486403 -2.008623 1.353536 -4.837431 4.991103 45.338573 1.801199 4.747960 -35.717758 -3.824305 5.977098 -23.867020 2.425
172 1.394668 8.218590 0.432578 -0.003181 31.384497 -1.119125 0.481357 9.746015 -0.681514 -0.232873 0.021375 0.736132 0.027272 -1.524101 0
.983109 0.228650 -5.678046 -0.046908 0.350175 3.762846 0.042326 0.146179 5.250207 -0.859184 -0.827984 -2.484071 0.268577 -0.541081 1.6046
91 -0.510899 -0.368184 -1.910566 -3.866541 0.679605 -6.550858 2.304412 -2.272054 -9.456560 -4.686183 3.395544 23.240416 1.942752 -0.98583
7 28.160833 1.395764 -4.232559 30.758883 1.774767 -4.438025 35.896500 1.299924 3.985591 -13.608424 -5.048252 3.397160 31.081030 0.172570
0.303916 -0.251996 1.000991 0.606965 1.831054 -0.171050 0.735471 1.315514 -0.734380 0.803324 1.898607 -0.874013 0.937502 1.095077 0.61809
2 -0.355409 -5.732562 0.710163 0.959974 0.932712 0.306670 -0.049563 5.357992

***Correct***

Time taken: 0.000394 seconds
```

[11] Parallel Algorithm 3 using MPI + openMP

I. Create an MPI environment.
II. Master Process:
    1. Input sparse matrix in CSR format.
    2. Create row offset array using prefix sum
    3. Input dense vector from file
    4. Divide the answer calculation into equal blocks and send required data(used to calculate answer) to child processes.
    5. Data passed: rowOffset, value(sub-array), b, startIndex, column(sub-array)
    6. Calculate partial answer for it's own sub-problem
    7. Receive the partial answer from the child processes.
    8. Add the partial answer to the final answer.
III. Child Process:
    1. Receive the data from the master process.
    2. Apply SPMV algorithm using openMP library using binary search [used in openmp_algo3] on partial data to calculate partial answer.
    3. Finally send the partial answer to the master process.
IV. Clear MPI environment.

Execution time using 4 processes and 1 thread: 0.000926 seconds
Execution time using 4 processes and 2 threads: 0.000714 seconds
Execution time using 4 processes and 4 threads: 0.000282 seconds
Execution time using 2 processes and 8 threads: 0.000402 seconds
Execution time using 2 processes and 16 threads: 0.000201 seconds

```
The final answer is:
0.755918 -0.325148 0.001030 1.262419 0.252252 -1.498094 -6.798784 2.392391 3.165552 -1.491856 -0.657184 -1.608070 0.378394 -0.748113 -0.0
15497 0.457518 0.670194 -13.746393 0.474922 0.539500 -7.601586 -0.069925 0.021011 1.341256 0.243773 -2.935911 5.199205 -2.845950 1.836988
 2.228165 1.818666 1.318007 -0.519863 0.223857 -0.567833 0.016591 2.026767 1.509110 13.869150 -2.720471 -1.666859 6.423777 -2.987484 5.41
0840 26.622616 -4.486403 -2.008623 1.353536 -4.837431 4.991103 45.338573 1.801199 4.747960 -35.717758 -3.824305 5.977098 -23.867020 2.425
172 1.394668 8.218590 0.432578 -0.003181 31.384497 -1.119125 0.481357 9.746015 -0.681514 -0.232873 0.021375 0.736132 0.027272 -1.524101 0
.983109 0.228650 -5.678046 -0.046908 0.350175 3.762846 0.042326 0.146179 5.250207 -0.859184 -0.827984 -2.484071 0.268577 -0.541081 1.6046
91 -0.510899 -0.368184 -1.910566 -3.866541 0.679605 -6.550858 2.304412 -2.272054 -9.456560 -4.686183 3.395544 23.240416 1.942752 -0.98583
7 28.160833 1.395764 -4.232559 30.758883 1.774767 -4.438025 35.896500 1.299924 3.985591 -13.608424 -5.048252 3.397160 31.081030 0.172570
0.303916 -0.251996 1.000991 0.606965 1.831054 -0.171050 0.735471 1.315514 -0.734380 0.803324 1.898607 -0.874013 0.937502 1.095077 0.61809
2 -0.355409 -5.732562 0.710163 0.959974 0.932712 0.306670 -0.049563 5.357992

The sequential answer is:
0.755918 -0.325148 0.001030 1.262419 0.252252 -1.498094 -6.798784 2.392391 3.165552 -1.491856 -0.657184 -1.608070 0.378394 -0.748113 -0.0
15497 0.457518 0.670194 -13.746393 0.474922 0.539500 -7.601586 -0.069925 0.021011 1.341256 0.243773 -2.935911 5.199205 -2.845950 1.836988
 2.228165 1.818666 1.318007 -0.519863 0.223857 -0.567833 0.016591 2.026767 1.509110 13.869150 -2.720471 -1.666859 6.423777 -2.987484 5.41
0840 26.622616 -4.486403 -2.008623 1.353536 -4.837431 4.991103 45.338573 1.801199 4.747960 -35.717758 -3.824305 5.977098 -23.867020 2.425
172 1.394668 8.218590 0.432578 -0.003181 31.384497 -1.119125 0.481357 9.746015 -0.681514 -0.232873 0.021375 0.736132 0.027272 -1.524101 0
.983109 0.228650 -5.678046 -0.046908 0.350175 3.762846 0.042326 0.146179 5.250207 -0.859184 -0.827984 -2.484071 0.268577 -0.541081 1.6046
91 -0.510899 -0.368184 -1.910566 -3.866541 0.679605 -6.550858 2.304412 -2.272054 -9.456560 -4.686183 3.395544 23.240416 1.942752 -0.98583
7 28.160833 1.395764 -4.232559 30.758883 1.774767 -4.438025 35.896500 1.299924 3.985591 -13.608424 -5.048252 3.397160 31.081030 0.172570
0.303916 -0.251996 1.000991 0.606965 1.831054 -0.171050 0.735471 1.315514 -0.734380 0.803324 1.898607 -0.874013 0.937502 1.095077 0.61809
2 -0.355409 -5.732562 0.710163 0.959974 0.932712 0.306670 -0.049563 5.357992

***Correct***

Time taken: 0.000402 seconds
```

[12] Parallel Algorithm 4 (using pthread along with Hillis Algorithm)

1. Convert sparse matrix into CSR format.
2. rowOffset array in CSR format is calculated using prefix sum method which is here calculated using Hillis Algorithm.
3. Iterate through every non zero element by dividing operations between threads using pthread to execute in parallel where each thread works on (number_of_non_zero_elements/ MAX_THREADS) non-zero elements.
4. Find the row number from rowOffset corresponding to the non-zero values using binary search (lower_bound).
5. Multiply non zero value with dense vector index corresponding to the column index of the input matrix.
6. Add the result to the row index found in step 3 in the resultant vector.

Results and Discussion

For sequential algorithm:
Execution time of the program: 0.001508 second.

For parallel algorithm 1:
Execution time using 1 thread: 0.000214 second.
Execution time using 2 threads: 0.000036 second.
Execution time using 4 threads: 0.000027 second.

Execution time using 8 threads: 0.000271 second.
Execution time using 16 threads: 0.001342 second.

Inference -
We can see that execution time decreases on increasing the number of threads initially. Later, the time is not decreasing due to communication overhead , idle time or contention. [3]

For parallel algorithm 2:
Execution time using 1 thread: 0.000299 second.
Execution time using 2 threads: 0.000285 second.
Execution time using 4 threads: 0.000124 second.
Execution time using 8 threads: 0.000102 second.
Execution time using 16 threads: 0.000132 second.

Inference -
We can see that execution time decreases on increasing the number of threads initially. Later, the time is not decreasing due to communication overhead, idle time or contention.

For parallel algorithm 3:
Execution time using 1 thread: 0.000198 second.
Execution time using 2 threads: 0.000185 second.
Execution time using 4 threads: 0.000127 second.
Execution time using 8 threads: 0.000112 second.
Execution time using 16 threads: 0.000134 second.

Inference -
We can see that execution time decreases on increasing the number of threads initially. Later, the time is not decreasing due to communication overhead,idle time or contention.

For parallel algorithm 3 using CUDA:
Time taken by program using 1 blocks and 1 threads is 0.379053 sec
Time taken by program using 2 blocks and 128 threads is 0.223105 sec
Time taken by program using 4 blocks and 2 threads is 0.277948 sec
Time taken by program using 4 blocks and 64 threads is 0.276365 sec
Time taken by program using 8 blocks and 32 threads is 0.277838 sec
Time taken by program using 16 blocks and 16 threads is 0.210272 sec

Inference -
We can see that execution time decreases on balancing the number of threads and number of blocks.

For parallel algorithm 3 using MPI:
Execution time using 4 processes: 0.000476 seconds
Execution time using 2 processes: 0.000442 seconds
Execution time using 1 process: 0.000470 seconds

Inference -
We can see that execution time has no significant difference on changing the number of processes as we are using a distributed system.

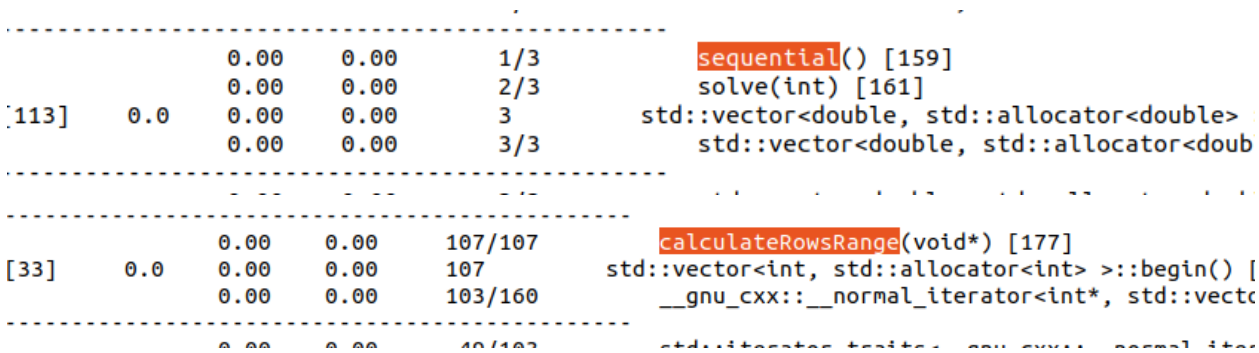
For parallel algorithm 3 using MPI + pthread:
Execution time using 4 processes and 1 thread: 0.000876 seconds
Execution time using 4 processes and 2 threads: 0.000394 seconds
Execution time using 4 processes and 4 threads: 0.000161 seconds
Execution time using 2 processes and 8 threads: 0.000219 seconds
Execution time using 2 processes and 16 threads: 0.000101 seconds

Inference -
We can see that execution time decreases on increasing the number of threads initially. Later, the time is not decreasing due to communication overhead, idle time or contention.

For parallel algorithm 3 using MPI + openMP:
Execution time using 4 processes and 1 thread: 0.000926 seconds
Execution time using 4 processes and 2 threads: 0.000714 seconds
Execution time using 4 processes and 4 threads: 0.000282 seconds
Execution time using 2 processes and 8 threads: 0.000402 seconds
Execution time using 2 processes and 16 threads: 0.000201 seconds

Inference -
We can see that execution time decreases on increasing the number of threads initially. Later, the time is not decreasing due to communication overhead, idle time or contention.

Profiling results (using gprof) for parallel algorithm 3 (binary search) -



Conclusion

As seen from results, parallel codes have lesser execution time as compared to the sequential code. While using pThread and openMP libraries we are doing multicore processing which in turn distributes the burden on each thread resulting in faster execution.
While using MPI we are using distributed computing i.e running a segment of problem on different systems. After parallelizing the code in each child machine, we have improved the program using threads.

Parallel Algorithm 1 has better load balancing than parallel algorithm 2.
Parallel Algorithm 3 is better than both 1 and 2 because we are operating on elements.
Parallel Algorithm 4 is the best algorithm. Here, the rowOffset is getting calculated using Hillis Algorithm. And further calculations are same as parallel algorithm 3.

References

1. https://www.sciencedirect.com/topics/computer-science/sparse-matrix-vector-multiplication
2. https://www.geeksforgeeks.org/sparse-matrix-representations-set-3-csr/
3. https://www.baeldung.com/cs/servers-threads-number