Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Science and Engineering (Data Science)

Subject: Machine Learning – IV Laboratory AY:
2024-25

Name: Jhanvi Parekh
Sap ID: 60009210033

Experiment 2

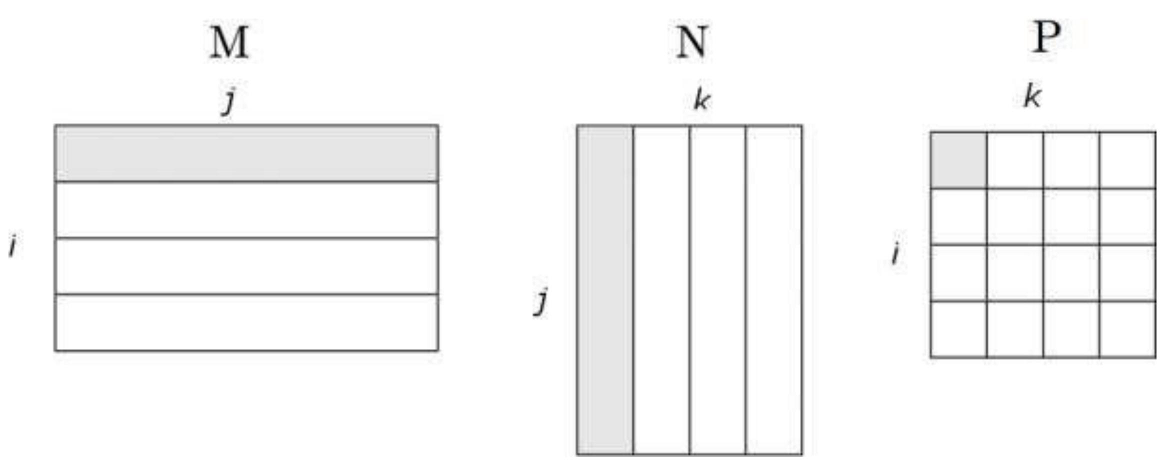
Aim: Implement program for Matrix Vector Multiplication using Map Reduce

Theory:

Matrix Multiplication

Matrix-vector and matrix-matrix calculations fit nicely into the MapReduce style of computing. In this post I will only examine matrix-matrix calculation as described in [1, ch.2].

Suppose we have a pxq matrix M, whose element in row i and column j will be denoted m_{ij} and a qxr matrix N whose element in row j and column k is donated by n_{jk} then the product P = MN will be pxr matrix P whose element in row i and column k will be donated by p_{ik}, where $P(i,k)$ = $m_{ij} * n_{jk}$.



Matrix Data Model for MapReduce

We represent matrix M as a $M(I, J, V)$ relation , with tuples (i, j, m_{ij}), and matrix N as a relation $N(J, K, W)$, with (j, k, n_{jk}) tuples . Most matrices are sparse so large amount of cells have value zero. When we represent matrices in this form, we do not need to keep entries for the cells that have values of zero to save large amount of disk space. As input data files, we store matrix M and N on HDFS in following format:

M, i, j, m_{ij}

M,0,0,10.0

M,0,2,9.0

M,0,3,9.0

M,1,0,1.0

M,1,1,3.0

M,1,2,18.0

M,1,3,25.2

....

N, j, k, n_{jk}

N,0,0,1.0

N,0,2,3.0

N,0,4,2.0

N,1,0,2.0

N,3,2,-1.0

N,3,6,4.0

N,4,6,5.0 N,4,0,-

1.0

....

MapReduce

We will write Map and Reduce functions to process input files. Map function will produce $key, value$ pairs from the input data as it is described in Algorithm 1. Reduce function uses the output of the Map function and performs the calculations and produces $key, value$ pairs as described in Algorithm 2. All outputs are written to HDFS.

Algorithm 1: The Map Function

for *each element* m_{ij} *of* M **do**
    produce $(key, value)$ pairs as $((i,k),(M,j,m_{ij}))$, for $k = 1,2,3,..$ up to the number of columns of N
for *each element* n_{jk} *of* N **do**
    produce $(key, value)$ pairs as $((i,k),(N,j,n_{jk}))$, for $i = 1,2,3,...$ up to the number of rows of M
return *Set of (key, value) pairs that each key,* (i,k)*, has a list with values* (M,j,m_{ij}) *and* (N,j,n_{jk}) *for all possible values of* j

Algorithm 2: The Reduce Function

for *each key (i,k)* **do**
    sort values begin with M by j in $list_M$
    sort values begin with N by j in $list_N$
    multiply m_{ij} and n_{jk} for j_{th} value of each list
    sum up $m_{ij} * n_{jk}$
return $(i,k), \sum_{j=1} m_{ij} * n_{jk}$

The value in row i and column k of product matrix P will be:

$$P_{(i,k)} = \sum_{j=1} m_{ij} * n_{jk}$$.

Let me examine the algorithms on an example to explain the algorithms better. Suppose we have two matrices, M, 2×3 matrix, and N, 3×2 matrix as follows:

The product P of MN will be as follows:

$$\begin{bmatrix} 1a + 2c + 3e & 1b + 2d + 3f \\ 4a + 5c + 6e & 4b + 5d + 6f \end{bmatrix}$$

Sample Problem:

Department of Computer Science and Engineering (Data Science)

(*) Matrix multiplication using Map-reduce:

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \\ 3 & 4 \end{bmatrix}_{i \times j} \qquad B = \begin{bmatrix} 1 & 2 \\ 1 & 3 \end{bmatrix}_{j \times k}$$

step 1: Input

Matrix	i/k	j	value
A	0	0	1
A	0	1	2
A	1	0	2
A	1	1	1
A	2	0	3
A	2	1	4
B	0	0	1
B	0	1	1
B	1	0	2
B	1	1	3

(<key, value>)

(*) important: (k,j) is in (col,row) format

step 2: Mapper = <key, value>

key (i/k)	value <Matrix_name, j, value>
0	(A,0,1)
0	(A,1,2)
1	(A,0,2)
1	(A,1,1)
2	(A,0,3)
2	(A,1,4)
0	(B,0,1)
0	(B,1,1)
1	(B,0,2)
1	(B,1,3)

{* imp (same as above)}

step 3: Reducer function → step 1 → shuffle

(i,k)	
(0,0)	(A,0,1), (A,1,2), (B,0,1), (B,1,1)
(0,1)	(A,0,1), (A,1,2), (B,0,2), (B,1,3)
(1,0)	(A,0,2), (A,1,1), (B,0,1), (B,1,1)
(1,1)	(A,0,2), (A,1,1), (B,0,2), (B,1,3)
(2,0)	(A,0,3), (A,1,4), (B,0,1), (B,1,1)
(2,1)	(A,0,3), (A,1,4), (B,0,2), (B,1,3)

step 4: Reducer function step 2 → reducing

(i,k)	
(0,0)	$[(1\times1) + (2\times1)] = 3$
(0,1)	$[(1\times2) + (2\times3)] = 8$
(1,0)	$[(2\times1) + (1\times1)] = 3$
(1,1)	$[(2\times2) + (1\times3)] = 7$
(2,0)	$[(3\times1) + (4\times1)] = 7$
(2,1)	$[(3\times2) + (4\times3)] = 18$

$$A \times B = \begin{bmatrix} 3 & 8 \\ 3 & 7 \\ 7 & 18 \end{bmatrix}$$

Lab Exercise:

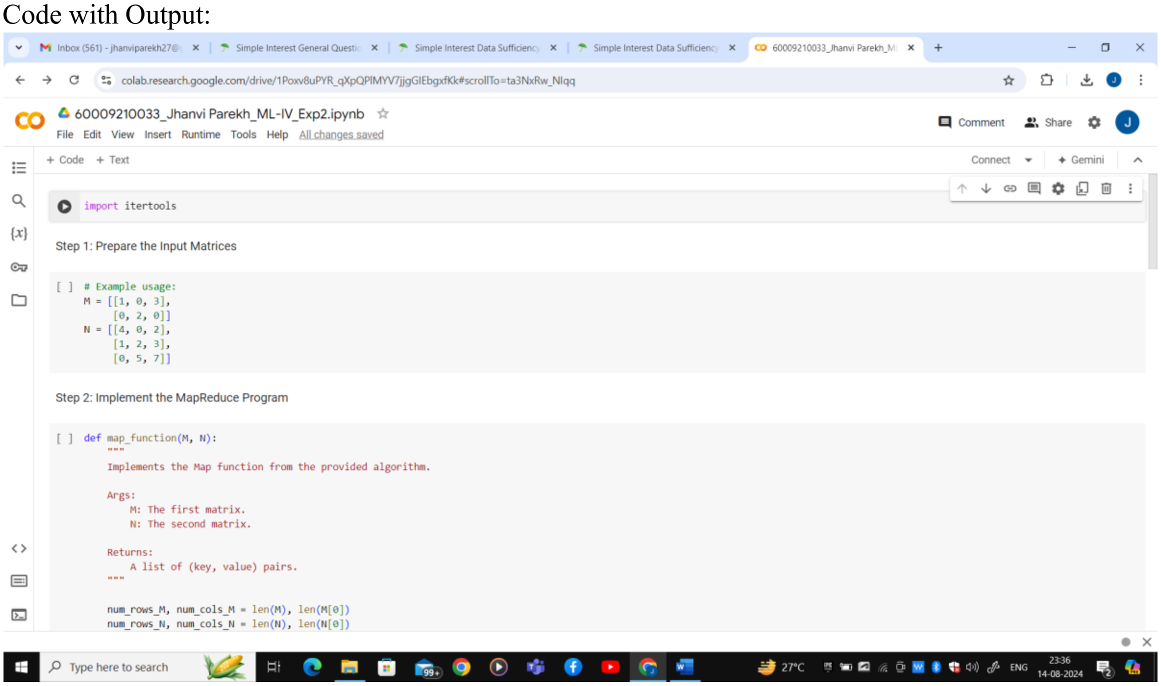
Step 1: Prepare two input matrices that you want to multiply. Ensure that the matrices are appropriately formatted and have compatible dimensions for multiplication.

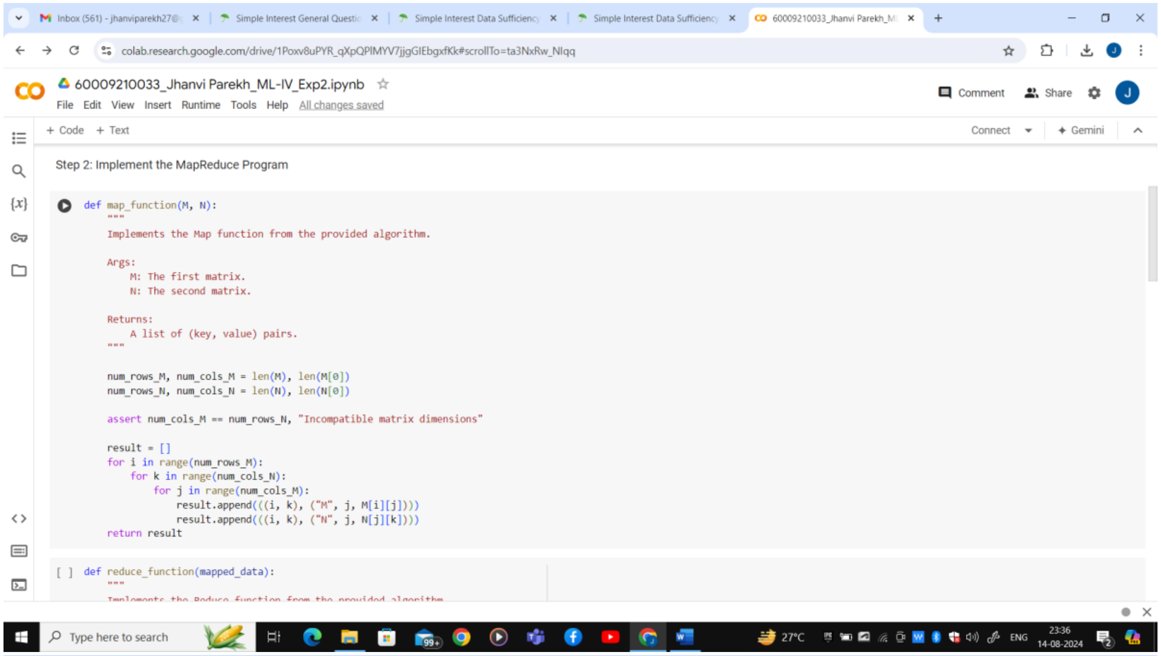
Step 2: Implement a MapReduce program to perform matrix multiplication. This program should consist of:

- A Mapper function that processes the cell values of the input matrices, emits key-value pairs (output cell location, product), and includes logic to determine the correct intermediate key for the product.
- A Reducer function that receives key-value pairs, groups them by key (output cell location), and performs the sum operation to get the final result for each cell.

[Along with matrix multiplication, carry out matrix addition using map-reduce provided time permits.]

Code with Output:




60009210033_Jhanvi Parekh_ML-IV_Exp2.ipynb
Step 2: Implement the MapReduce Program
def map_function(M, N):
    """
    Implements the Map function from the provided algorithm.

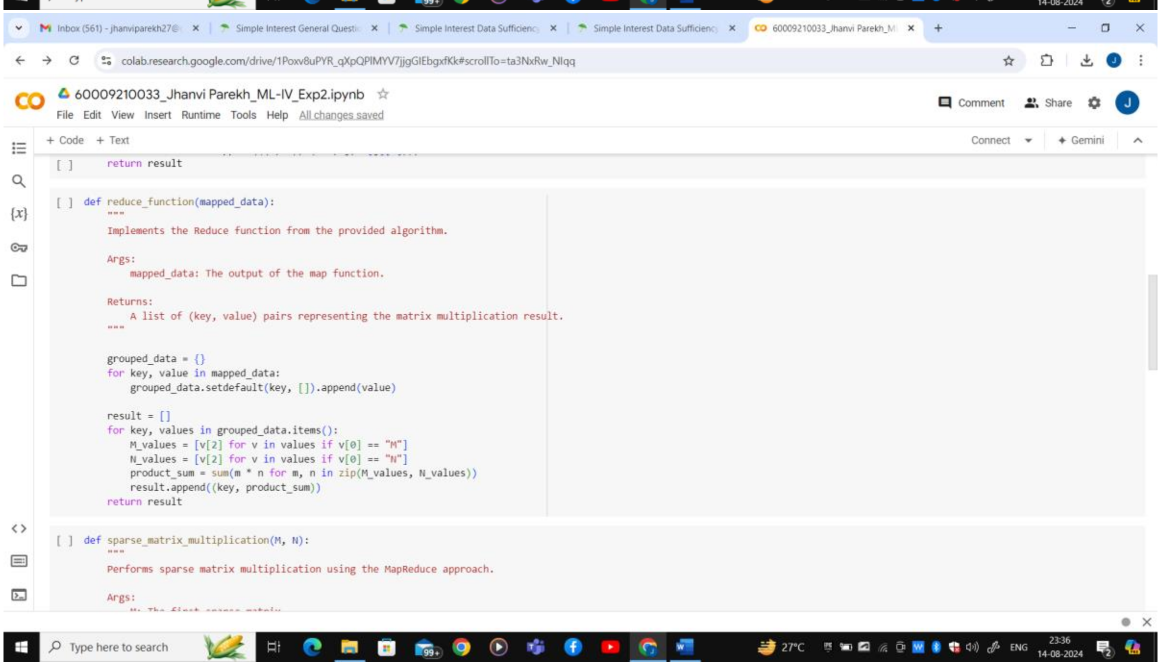
    Args:
        M: The first matrix.
        N: The second matrix.

    Returns:
        A list of (key, value) pairs.
    """

    num_rows_M, num_cols_M = len(M), len(M[0])
    num_rows_N, num_cols_N = len(N), len(N[0])

    assert num_cols_M == num_rows_N, "Incompatible matrix dimensions"

    result = []
    for i in range(num_rows_M):
        for k in range(num_cols_N):
            for j in range(num_cols_M):
                result.append(((i, k), ("M", j, M[i][j])))
                result.append(((i, k), ("N", j, N[j][k])))
    return result
def reduce_function(mapped_data):
    """


60009210033_Jhanvi Parekh_ML-IV_Exp2.ipynb
    return result
def reduce_function(mapped_data):
    """
    Implements the Reduce function from the provided algorithm.

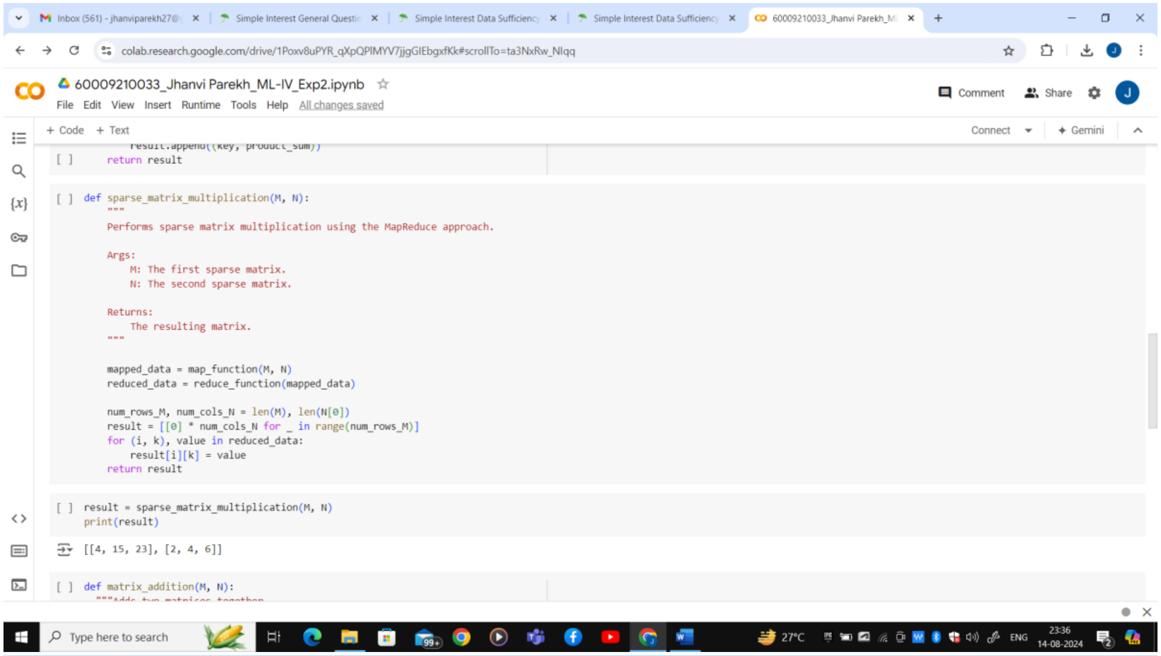
    Args:
        mapped_data: The output of the map function.

    Returns:
        A list of (key, value) pairs representing the matrix multiplication result.
    """

    grouped_data = {}
    for key, value in mapped_data:
        grouped_data.setdefault(key, []).append(value)

    result = []
    for key, values in grouped_data.items():
        M_values = [v[2] for v in values if v[0] == "M"]
        N_values = [v[2] for v in values if v[0] == "N"]
        product_sum = sum(m * n for m, n in zip(M_values, N_values))
        result.append((key, product_sum))
    return result
def sparse_matrix_multiplication(M, N):
    """
    Performs sparse matrix multiplication using the MapReduce approach.

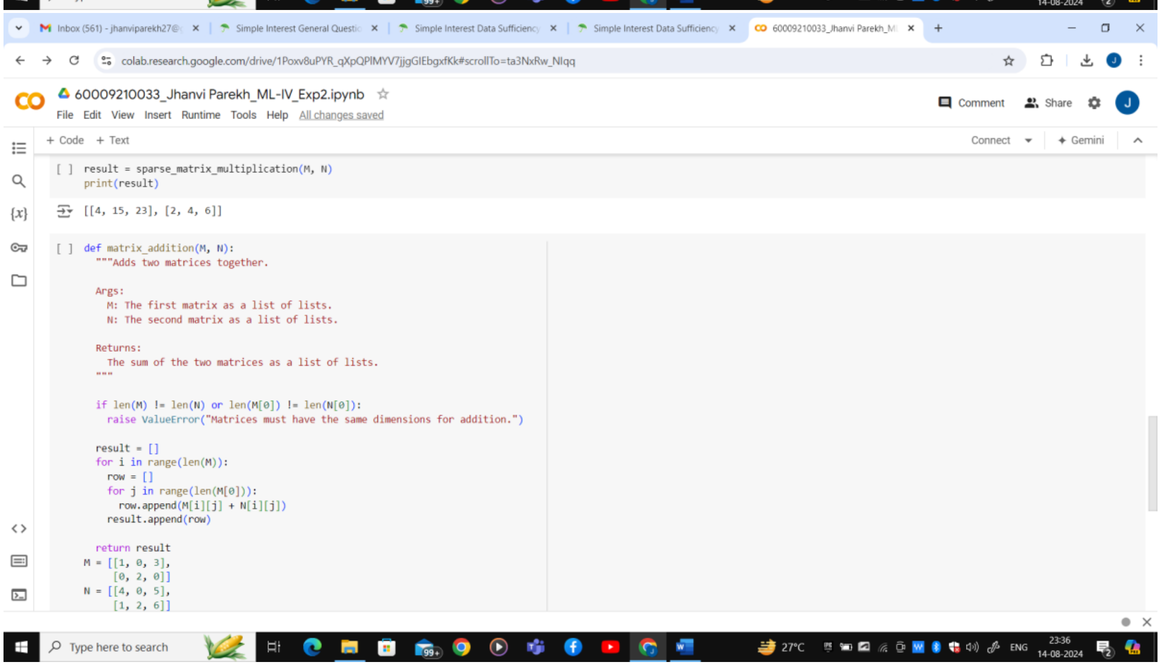
    Args:



```python
    return result

def sparse_matrix_multiplication(M, N):
    """
    Performs sparse matrix multiplication using the MapReduce approach.

    Args:
        M: The first sparse matrix.
        N: The second sparse matrix.

    Returns:
        The resulting matrix.
    """

    mapped_data = map_function(M, N)
    reduced_data = reduce_function(mapped_data)

    num_rows_M, num_cols_N = len(M), len(N[0])
    result = [[0] * num_cols_N for _ in range(num_rows_M)]
    for (i, k), value in reduced_data:
        result[i][k] = value
    return result

result = sparse_matrix_multiplication(M, N)
print(result)
```

```
[[4, 15, 23], [2, 4, 6]]
```

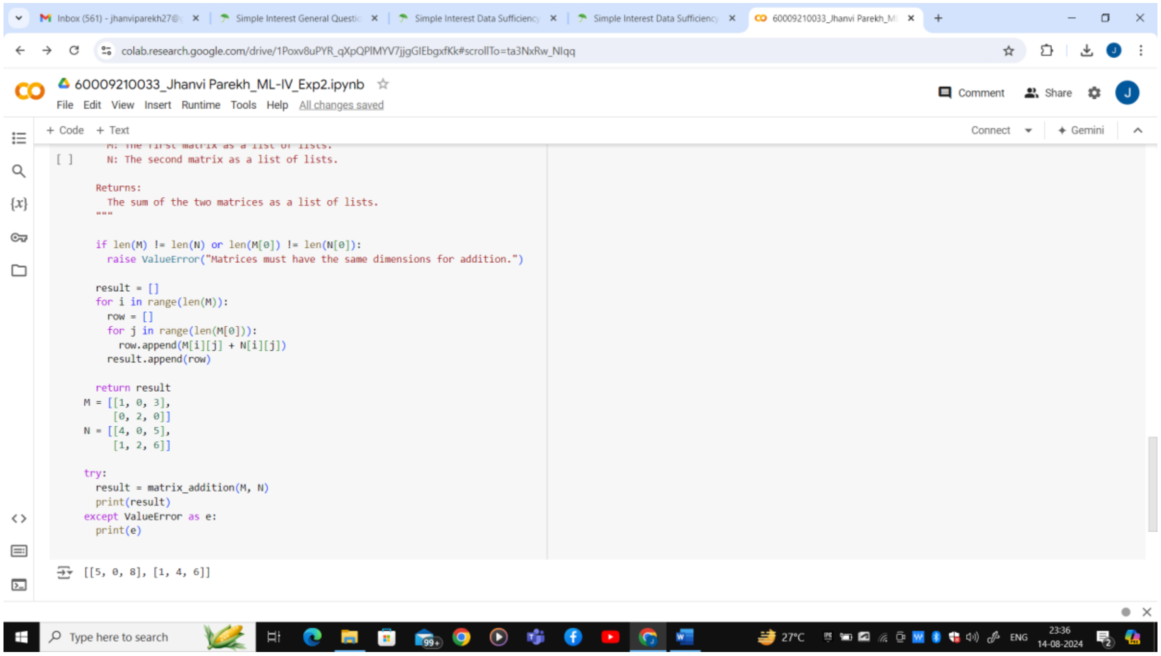
```python
def matrix_addition(M, N):
```



```python
result = sparse_matrix_multiplication(M, N)
print(result)
```

```
[[4, 15, 23], [2, 4, 6]]
```

```python
def matrix_addition(M, N):
  """Adds two matrices together.

  Args:
    M: The first matrix as a list of lists.
    N: The second matrix as a list of lists.

  Returns:
    The sum of the two matrices as a list of lists.
  """

  if len(M) != len(N) or len(M[0]) != len(N[0]):
    raise ValueError("Matrices must have the same dimensions for addition.")

  result = []
  for i in range(len(M)):
    row = []
    for j in range(len(M[0])):
      row.append(M[i][j] + N[i][j])
    result.append(row)

  return result
M = [[1, 0, 3],
     [0, 2, 0]]
N = [[4, 0, 5],
     [1, 2, 6]]
```



Link:
https://colab.research.google.com/drive/1Poxv8uPYR_qXpQPlMYV7jjgGIEbgxfKk?usp=sharing

Conclusion:
Thus, this experiment introduced matrix multiplication using MapReduce. We explored how to distribute this complex computational task across multiple nodes, demonstrating the scalability of the MapReduce framework. The experiment reinforced our understanding of parallel processing and data transformations on a distributed scale.