Assignment - 11

1. Describe Canon's Matrix Multiplication algorithm.
2. Implement Canon's Matrix Multiplication using collective communication.
3. Analyze the efficiency of the code.

- Canon's Matrix Multiplication is an algorithm for multiplying matrices it helps with to distribute the computation across multiple processors in a parallel or distributed computing environment it is useful when dealing with very large matrices that cannot be efficiently handled by a single processor.
- Canon's algorithm works as follows

1. Divide each matrix into submatrices.
2. Distribute these submatrices across the processors in a grid-like fashion.
3. Each processor computes the partial products of its assigned submatrices iteratively shift the submatrices horizontally and vertically, so that each processor multiplies its submatrices with the corresponding ones from neighboring processors.
4. Repeat the shifting and multiplication steps until each processor has performed all necessary multiplications.
5. Accumulate the partial results to obtain the final product matrix.

In [1]:
```
from mpi4py import MPI
import numpy as np
import time
import matplotlib.pyplot as plt
```

In [2]:
```
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

N = 2000
if N % size != 0:
    raise ValueError("Matrix size N must be divisible by the number of processes (size)")

block_size = N // size

print(f"Rank {rank}: Starting execution")

if rank == 0:
    print(f"Rank {rank}: Generating matrices A and B")
    A = np.random.randint(0, 10, (N, N))
    B = np.random.randint(0, 10, (N, N))
    print(f"Rank {rank}: Matrices A and B generated")
else:
    A = None
    B = None

print(f"Rank {rank}: Broadcasting matrices A and B")
start_time = time.time()
A = comm.bcast(A, root=0)
B = comm.bcast(B, root=0)
end_time = time.time()
print(f"Rank {rank}: Matrices A and B broadcasted")

A_rows = np.zeros((block_size, N), dtype=int)
comm.Scatter(A, A_rows, root=0)

start_time_multiplication = time.time()
C_rows = np.dot(A_rows, B)
end_time_multiplication = time.time()

C = None
if rank == 0:
    C = np.zeros((N, N), dtype=int)

comm.Gather(C_rows, C, root=0)

if rank == 0:
    print("Resultant Matrix C:")
    print(C)
    print("Broadcasting time:", end_time - start_time, "seconds")
    print("Matrix multiplication time:", end_time_multiplication - start_time_multiplicatio
```

```
Rank 0: Starting execution
Rank 0: Generating matrices A and B
Rank 0: Matrices A and B generated
Rank 0: Broadcasting matrices A and B
Rank 0: Matrices A and B broadcasted
Resultant Matrix C:
[[38924 38988 40084 ... 39540 39911 40176]
 [39181 39193 39492 ... 39247 39309 40381]
 [39406 39285 39590 ... 39814 39647 39856]
 ...
 [39500 40874 40270 ... 40317 40574 40609]
 [40338 41030 41022 ... 40296 41134 41550]
 [38284 38920 39233 ... 38895 38968 38838]]
Broadcasting time: 0.03937983512878418 seconds
Matrix multiplication time: 42.12184238433838 seconds
```

In [3]:
```
!mpiexec -n 4 python MPI_Scatter_Gather.py
```

```
Rank 3: Starting execution
Rank 3: Broadcasting matrices A and B
Rank 3: Matrices A and B broadcasted
Rank 1: Starting execution
Rank 1: Broadcasting matrices A and B
Rank 1: Matrices A and B broadcasted
Rank 2: Starting execution
Rank 2: Broadcasting matrices A and B
Rank 2: Matrices A and B broadcasted
Rank 0: Starting execution
Rank 0: Generating matrices A and B
Rank 0: Matrices A and B generated
Rank 0: Broadcasting matrices A and B
Rank 0: Matrices A and B broadcasted
Resultant Matrix C:
[[42017 41045 42732 ... 41943 40837 42255]
 [40844 40173 40798 ... 41069 39922 39766]
 [40892 39590 40565 ... 40460 39376 39704]
 ...
 [40474 40181 40543 ... 40339 39739 40542]
 [41234 40700 41083 ... 40461 40372 39411]
 [41092 40366 41294 ... 40061 39890 39640]]
Broadcasting time: 0.060237884521484375 seconds
Matrix multiplication time: 11.654186248779297 seconds
```

In [5]:
```
!mpiexec -n 8 python MPI_Scatter_Gather.py
```

```
Rank 7: Starting execution
Rank 7: Broadcasting matrices A and B
Rank 7: Matrices A and B broadcasted
Rank 3: Starting execution
Rank 3: Broadcasting matrices A and B
Rank 3: Matrices A and B broadcasted
Rank 5: Starting execution
Rank 5: Broadcasting matrices A and B
Rank 5: Matrices A and B broadcasted
Rank 6: Starting execution
Rank 6: Broadcasting matrices A and B
Rank 6: Matrices A and B broadcasted
Rank 1: Starting execution
Rank 1: Broadcasting matrices A and B
Rank 1: Matrices A and B broadcasted
Rank 2: Starting execution
Rank 2: Broadcasting matrices A and B
Rank 2: Matrices A and B broadcasted
Rank 0: Starting execution
Rank 0: Generating matrices A and B
Rank 0: Matrices A and B generated
Rank 0: Broadcasting matrices A and B
Rank 0: Matrices A and B broadcasted
Resultant Matrix C:
[[40325 40191 41154 ... 40507 40363 39973]
 [39863 39909 41875 ... 41396 39875 40330]
 [40414 39831 40746 ... 40748 40352 39065]
 ...
 [39360 40292 41247 ... 40491 40832 40227]
 [41656 41171 43231 ... 42439 41473 41172]
 [39526 38647 40187 ... 39084 38608 38932]]
Broadcasting time: 0.10047459602355957 seconds
Matrix multiplication time: 10.563331127166748 seconds
Rank 4: Starting execution
Rank 4: Broadcasting matrices A and B
Rank 4: Matrices A and B broadcasted
```

In [4]:
```python
from mpi4py import MPI
import numpy as np
import time

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

N = 2000
if N % size != 0:
    raise ValueError("Matrix size N must be divisible by the number of processes (size)")

block_size = N // size

print(f"Rank {rank}: Starting execution")

if rank == 0:
    print(f"Rank {rank}: Generating matrices A and B")
    A = np.random.randint(0, 10, (N, N))
    B = np.random.randint(0, 10, (N, N))
    print(f"Rank {rank}: Matrices A and B generated")
else:
    A = None
    B = None

print(f"Rank {rank}: Broadcasting matrices A and B")
start_time = time.time()
A = comm.bcast(A, root=0)
B = comm.bcast(B, root=0)
end_time = time.time()
print(f"Rank {rank}: Matrices A and B broadcasted")

A_rows = np.zeros((block_size, N), dtype=int)
comm.Scatter(A, A_rows, root=0)

start_time_multiplication = time.time()
C_rows = np.dot(A_rows, B)
end_time_multiplication = time.time()

start_time_gather = time.time()
C_all = np.zeros((N, N), dtype=int)
comm.Allgather(C_rows, C_all)
end_time_gather = time.time()

if rank == 0:
    print("Resultant Matrix C:")
    print(C_all)
    print("Broadcasting time:", end_time - start_time, "seconds")
    print("Gathering time:", end_time_gather - start_time_gather, "seconds")
    print("Matrix multiplication time:", end_time_multiplication - start_time_multiplicatio
```

```
Rank 0: Starting execution
Rank 0: Generating matrices A and B
Rank 0: Matrices A and B generated
Rank 0: Broadcasting matrices A and B
Rank 0: Matrices A and B broadcasted
Resultant Matrix C:
[[40916 40038 41686 ... 40639 42268 40762]
 [39408 38656 40396 ... 39191 40604 39282]
 [40379 39155 41555 ... 40121 41890 39871]
 ...
 [41187 40173 41164 ... 41221 41924 40741]
 [41633 40312 42298 ... 41181 42086 40899]
 [39550 38358 40014 ... 40075 40537 39391]]
Broadcasting time: 0.016689777374267578 seconds
Gathering time: 0.002426624298095703 seconds
Matrix multiplication time: 42.118860960006714 seconds
```

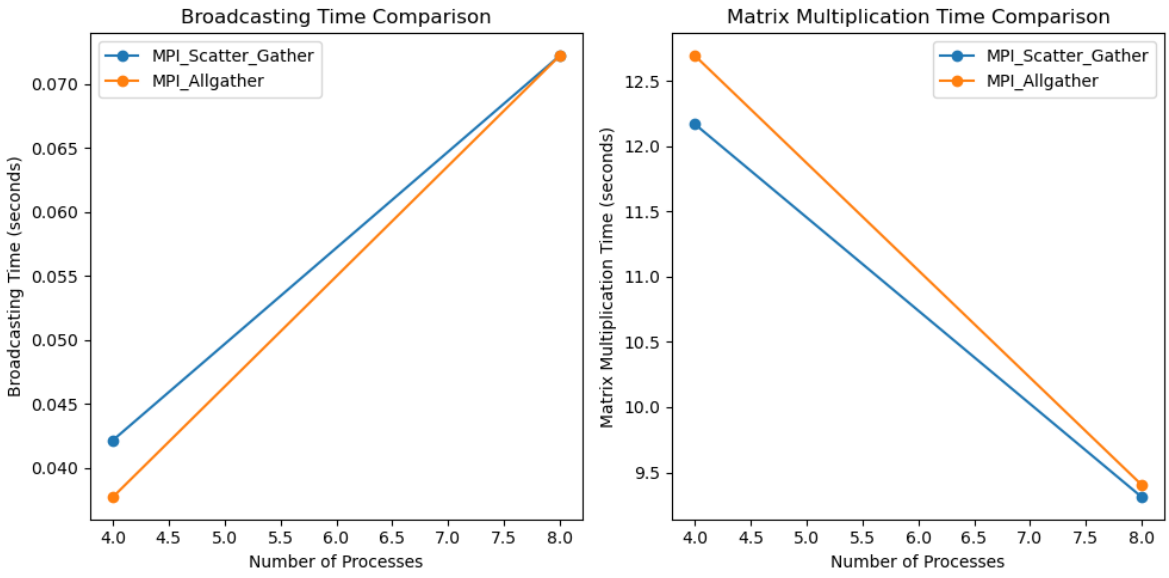
In [6]:
```
!mpiexec -n 4 python MPI_Allgather.py
```

```
Rank 1: Starting execution
Rank 1: Broadcasting matrices A and B
Rank 1: Matrices A and B broadcasted
Rank 2: Starting execution
Rank 2: Broadcasting matrices A and B
Rank 2: Matrices A and B broadcasted
Rank 3: Starting execution
Rank 3: Broadcasting matrices A and B
Rank 3: Matrices A and B broadcasted
Rank 0: Starting execution
Rank 0: Generating matrices A and B
Rank 0: Matrices A and B generated
Rank 0: Broadcasting matrices A and B
Rank 0: Matrices A and B broadcasted
Resultant Matrix C:
[[41981 42298 42010 ... 41927 40632 40686]
 [40844 42040 41133 ... 41613 40279 41009]
 [40145 40810 39721 ... 40101 39097 39790]
 ...
 [40915 40933 40913 ... 40927 39648 40175]
 [40807 41294 40486 ... 41782 40666 40397]
 [40110 40841 40445 ... 41008 40574 40400]]
Broadcasting time: 0.03273797035217285 seconds
Gathering time: 0.01018381118774414 seconds
Matrix multiplication time: 11.341147899627686 seconds
```

In [7]:
```
!mpiexec -n 8 python MPI_Allgather.py
```

```
Rank 5: Starting execution
Rank 5: Broadcasting matrices A and B
Rank 5: Matrices A and B broadcasted
Rank 6: Starting execution
Rank 6: Broadcasting matrices A and B
Rank 6: Matrices A and B broadcasted
Rank 7: Starting execution
Rank 7: Broadcasting matrices A and B
Rank 7: Matrices A and B broadcasted
Rank 4: Starting execution
Rank 4: Broadcasting matrices A and B
Rank 4: Matrices A and B broadcasted
Rank 0: Starting execution
Rank 0: Generating matrices A and B
Rank 0: Matrices A and B generated
Rank 0: Broadcasting matrices A and B
Rank 0: Matrices A and B broadcasted
Resultant Matrix C:
[[38826 38685 39926 ... 40178 39925 38546]
 [41287 40079 40148 ... 41615 41193 41219]
 [39324 39803 40427 ... 40839 40679 40418]
 ...
 [39700 39258 40242 ... 40461 39879 39241]
 [39591 39742 40000 ... 40148 39897 40145]
 [40836 40086 40435 ... 41049 40958 40317]]
Broadcasting time: 0.09126925468444824 seconds
Gathering time: 0.2480170726776123 seconds
Matrix multiplication time: 10.671711921691895 seconds
Rank 1: Starting execution
Rank 1: Broadcasting matrices A and B
Rank 1: Matrices A and B broadcasted
Rank 2: Starting execution
Rank 2: Broadcasting matrices A and B
Rank 2: Matrices A and B broadcasted
Rank 3: Starting execution
Rank 3: Broadcasting matrices A and B
Rank 3: Matrices A and B broadcasted
```

In [8]:
```
scatter_gather_processes = [4, 8]
scatter_gather_broadcasting_time = [0.04213356971740723, 0.0722498893737793]
scatter_gather_multiplication_time = [12.16959547996521, 9.310021162033081]

allgather_processes = [4, 8]
allgather_broadcasting_time = [0.037689924240112305, 0.0722506046295166 ]
allgather_multiplication_time = [12.694447040557861, 9.402881145477295]

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.plot(scatter_gather_processes, scatter_gather_broadcasting_time, marker='o', label='MPI
plt.plot(allgather_processes, allgather_broadcasting_time, marker='o', label='MPI_Allgather
plt.xlabel('Number of Processes')
plt.ylabel('Broadcasting Time (seconds)')
plt.title('Broadcasting Time Comparison')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(scatter_gather_processes, scatter_gather_multiplication_time, marker='o', label='M
plt.plot(allgather_processes, allgather_multiplication_time, marker='o', label='MPI_Allgath
plt.xlabel('Number of Processes')
plt.ylabel('Matrix Multiplication Time (seconds)')
plt.title('Matrix Multiplication Time Comparison')
plt.legend()

plt.tight_layout()
plt.show()
```



1. MPI_Scatter_Gather:

- With 4 processes, the broadcasting time is lower compared to MPI_Allgather, but the matrix multiplication time is higher.
- With 8 processes, the broadcasting time increases slightly, but the matrix multiplication time decreases significantly compared to 4 processes.

2. MPI_Allgather:

- With 4 processes, the broadcasting time is slightly higher compared to MPI_Scatter_Gather, but the gathering time is introduced. However, the matrix multiplication time is comparable to MPI_Scatter_Gather.
- With 8 processes, the broadcasting time is slightly higher compared to MPI_Scatter_Gather, and the gathering time becomes noticeable. However, the matrix multiplication time is the lowest among all configurations, indicating better parallel efficiency.

3. Conclusion:

- MPI_Scatter_Gather might be more efficient for smaller-scale parallelization, where the gathering is not significant compared to the reduction in matrix multiplication time.

- MPI_Allgather becomes more efficient as the number of processes increases, especially for larger-scale parallelization, due to its better load balancing and reduced communication-to-computation ratio.