Distributed Systems Project

Team 61 - Malla Sailesh , Nikunj Garg

1. Objective

The provided code implements a distributed version of the **Floyd-Warshall algorithm** using the Message Passing Interface (MPI). The goal is to calculate the shortest paths between all pairs of vertices in a directed graph. The code also handles negative weight edges and detects the presence of negative cycles in the graph.

2. Overview of the Algorithm

The Floyd-Warshall algorithm is a dynamic programming technique for solving the APSP problem. It works by iteratively updating the shortest path between all pairs of vertices using an intermediate vertex. The algorithm runs in O(n^3), where n is the number of vertices.

3. Parallelization Approach

The program divides the workload among multiple processes using MPI. Each process is responsible for computing a subset of the rows of the adjacency matrix. Here's how it works:

1. **Matrix Initialization**:
    - Process 0 initializes the adjacency matrix. For diagonal elements, the weight is set to 0 (distance from a vertex to itself). Other elements are initialized to infinity (`INF`) to represent no path.
    - For input edges, the weight is updated to the smaller of the given weight and the existing value (to handle multiple edges between two vertices).
2. **Matrix Distribution**:
    - The adjacency matrix is distributed row-wise to processes using `MPI_Scatterv`. Each process receives a block of rows based on the number of rows it owns.

`MPI_Scatterv(matrix.data(), send_counts.data(), displs.data(), MPI_LONG_LONG_INT, local_matrix.data(), send_counts[rank], MPI_LONG_LONG_INT, 0, MPI_COMM_WORLD);` - `send_counts` and `displacement` defines how much data each process receives and its offset in the matrix.

3. **Core Algorithm**:
   - The algorithm iterates over all vertices k (acting as intermediate vertices).
   - The process owning the k-th row broadcasts it to all other processes. This is found using this formula :-

```
int owner = (k < extra*(rows_per_process+1)) ?
k/(rows_per_process+1):
((k-extra*(rows_per_process+1))/(rows_per_process))+extra
```

   - Each process updates its local rows by considering the kkk-th row and applying the relaxation formula:

```
if(local_matrix[i*n+k] != INF && row_k[j] != INF &&
local_matrix[i*n + j] > local_matrix[i*n+k] + row_k[j]){

    local_matrix[i*n + j] = local_matrix[i*n + k] + row_k[j];
}
```

   - This ensures that the shortest paths through vertex k are computed for all vertex pairs.
4. **Negative Cycle Detection**:
   - After computing the shortest paths, the algorithm checks for negative cycles. by verifying if the distance can still be reduced after n iterations.

```
    if(local_matrix[i*n+k] != INF && row_k[j] != INF &&
local_matrix[i*n + j] > local_matrix[i*n+k] + row_k[j]){

            // Negative Cycle exists

        }
```

5. **Result Gathering**:
   - The updated rows are gathered back to Process 0 using `MPI_Gatherv`.

```
MPI_Gatherv(local_matrix.data(), send_counts[rank],
MPI_LONG_LONG_INT, matrix.data(), send_counts.data(), displs.data(),
MPI_LONG_LONG_INT, 0, MPI_COMM_WORLD);
```

4. Performance Considerations

- **Parallel Efficiency**:
  - The matrix is distributed across processes, reducing the computational workload per process.
  - Communication overhead arises from broadcasting rows and gathering results.
- **Load Balancing**:
  - Rows are evenly distributed using `rows_per_process` and `extra` calculations, ensuring fair workload distribution.

    ```
    rows_per_process = n / p;      extra = n%p;
    ```

  - Starting and ending row for a process can be found using the below formulas:-

    ```
    int sr = rows_per_process*rank + min(rank, extra);

    int er = sr + rows_per_process + (rank < extra? 0: -1);
    ```

- **Negative Cycle Detection**:
  - Performed after the main computation, with minimal additional communication overhead to detect if there are negative cycles in the graph by verifying if the distance can still be reduced after n iterations.

```
if(local_matrix[i*n+k] != INF && row_k[j] != INF && local_matrix[i*n
+ j] > local_matrix[i*n+k] + row_k[j]){

                // Negative Cycle

                break;

            }
```

5. Analysis

The sequential Floyd's algorithm has a time complexity of O(n3). For the parallel implementation, the complexity is influenced by computation and communication costs:

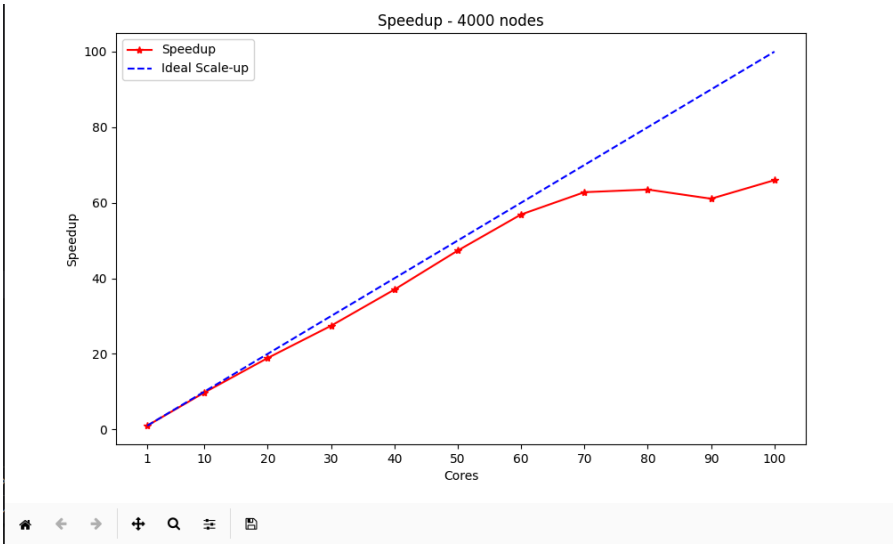
1. **Computation Complexity:**
   Each process updates at most n/p rows of the matrix, with each row requiring O(n) operations per iteration. Thus, the computational complexity of the parallel inner two loops is O(n^2/p) and overall complexity is O(n^3/p) for algorithm.
2. **Communication Complexity:**
   Broadcasting a row to p processes involvesO(logp) message-passing steps, each taking O(n) time. For n iterations, the communication complexity is O(n · nlogp) = O(n^2 * logp).

Combining both, the overall complexity becomes O(n^3/p + n^2*logp) of algorith,/

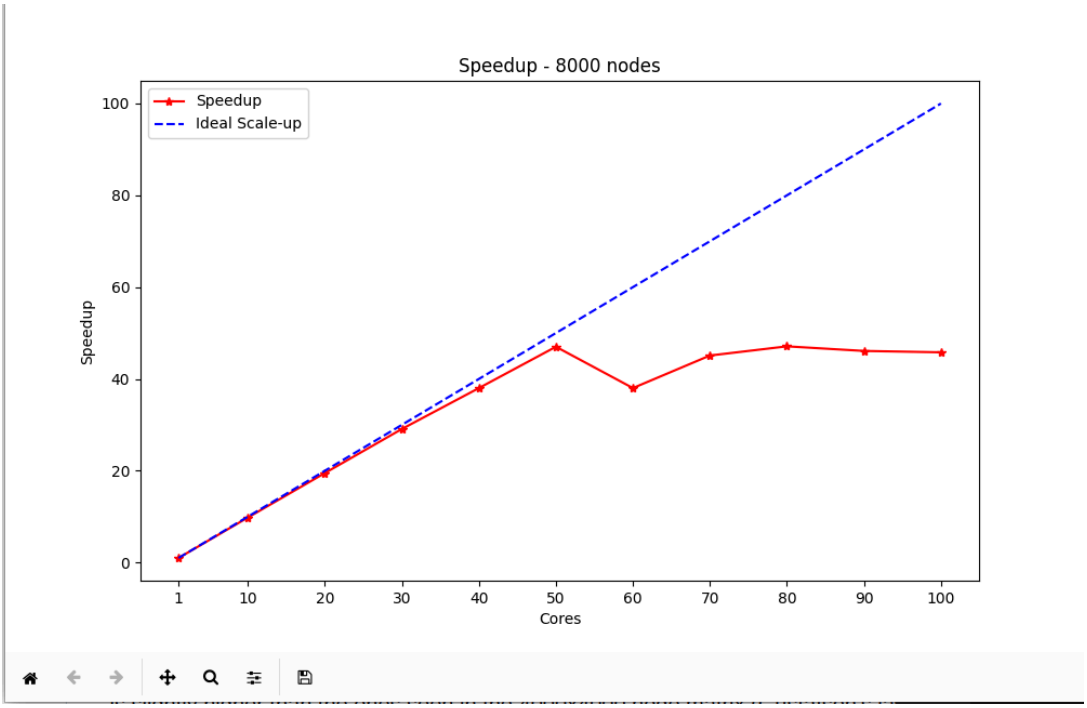
6. Plot

To substantiate the correctness and performance of the implementation, tests were run with different number of processes. The running time of the algorithm is the factor determining the efficiency of this algorithm. observations have been analyzed using graphs.

The below graph shows the speedup of the algorithm for different number of cores using a 4000 node input dense matrix. Here, we can see that the speedup is close to linear for configurations upto 50 cores and then its gradually drops (**Amdahl's law,** Strong scaling - holds here as we are able achieve good speedup only upto a certain configuration and thereafter the speedup reduces).



The below graph shows the speedup of the algorithm for different number of processes using a 8000 node input dense matrix.



The below graph shows the speedup of the algorithm for different number of processes using a 30000 node input dense matrix. Speedup is slightly higher than the ones seen in the 4000 node matrix (**Gustafson's law**, Weak scaling - holds here as we are increasing the input data size and achieving better speedup).