Section 1

Execution

The code can be executed with ./run.sh <num_time_steps>

Section 2

Explanation of the code

The main code is in `src.c` which performs one run of message parsing by 3 different ways. Following is the general schema for each run:

1. Initialize data as a dynamic (but contiguous) block of size N∗N with random **double** in the range $[0, 1]$.
2. Repeat the following steps num_time_steps times:
   - Compute the stencil for the data.
   - Send/Recv in all possible directions by a specific methodology.
   - Compute the final halo (only the boundaries).
3. Free the data, and related data structures.

Schema 1: General Structure of the code

```
double stime = MPI_Wtime();
for(int t=0; t<num_time_steps; t++){
    // Perform stencil computation
    compute_stencil(data, side_len);

    // Send to possible directions
    if(can_transfer('u', my_rank, cluster_len, p)){
        // If it is possible to transfer upwards, send.
    }
    if(can_transfer('d', my_rank, cluster_len, p)){
        // If it is possible to transfer downwards, send.
    }
    if(can_transfer('l', my_rank, cluster_len, p)){
        // If it is possible to transfer leftwards, send.
    }
    if(can_transfer('r', my_rank, cluster_len, p)){
        /// If it is possible to transfer rightwards, send.
    }

    // Recieve from possible directions
    if(can_transfer('u', my_rank, cluster_len, p)){
        // If it is possible to transfer upwards, receive.
    }
    if(can_transfer('d', my_rank, cluster_len, p)){
        // If it is possible to transfer downwards, receive.
    }
    if(can_transfer('l', my_rank, cluster_len, p)){
       // If it is possible to transfer leftwards, receive.
    }
```

```
    if(can_transfer('r', my_rank, cluster_len, p)){
        // If it is possible to transfer rightwards, receive.
    }

    // Get the final averages for time t
    compute_halo(data, recv_data, side_len, my_rank, cluster_len, p);
}
double ftime = MPI_Wtime();

double time = ftime - stime;
MPI_Reduce (&time, &maxTime, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

// print only by rank 0
if (!my_rank) printf ("%lf\n", maxTime);
```

Following the above schema, the three strategies for message send/receive can be implemented inside each of the **if** condition. Following are three excerpts from our code for the three given strategies.

Part 1: Single MPI_Send/Recv

```
// sending upwards
if(can_transfer('u', my_rank, cluster_len, p)){
    int target = my_rank - cluster_len;
    for(int i=0; i<side_len; i++){
        MPI_Send(&data[0*side_len + i], 1, MPI_DOUBLE, target, i, MPI_COMM_WORLD);
    }
}

// recv from below
if(can_transfer('d', my_rank, cluster_len, p)){
    int source = my_rank + cluster_len;
    MPI_Status st;
    for(int i=0; i<side_len; i++){
        MPI_Recv(&recv_data[1][i], 1, MPI_DOUBLE, source, i, MPI_COMM_WORLD, &st);
    }
}
```

Here we need to make side_len number of MPI_Send() MPI_Recv() calls for transfer in a single direction.

Part 2: MPI_Send/Recv using MPI_Pack/Unpack

```
// sending upwards
if(can_transfer('u', my_rank, cluster_len, p)){
    int target = my_rank - cluster_len, upos = 0;
    for(int j=0; j<side_len; j++){
        MPI_Pack (&data[0*side_len + j], 1, MPI_DOUBLE, s_ubuf, side_len*8, &upos, MPI_COMM_WORLD);
    }
    MPI_Send(s_ubuf, side_len*8, MPI_PACKED, target, 1, MPI_COMM_WORLD);
}

// recv from below
if(can_transfer('d', my_rank, cluster_len, p)){
    int source = my_rank + cluster_len, curr_pos = 0;
    MPI_Status st;
    MPI_Recv(r_dbuf, side_len*8, MPI_PACKED, source, 1, MPI_COMM_WORLD, &st);
```

```
    MPI_Unpack(r_dbuf, side_len*8, &curr_pos, recv_data[1], side_len, MPI_DOUBLE, MPI_COMM_WORLD);
}
```

Here we need to make a single MPI_Send() MPI_Recv() call for transfer in a single direction with side_len number of MPI_Pack()/MPI_Unpack().

Part 3: MPI_Send/Recv using MPI_Datatype Vector

```
// define datatypes for row and col
MPI_Datatype row_vector, col_vector;
MPI_Type_vector (side_len, 1, 1, MPI_DOUBLE, &row_vector);
MPI_Type_commit (&row_vector);
MPI_Type_vector (side_len, 1, side_len, MPI_DOUBLE, &col_vector);
MPI_Type_commit (&col_vector);

// sending upwards (same for row and col)
if(can_transfer('u', my_rank, cluster_len, p)){
    int target = my_rank - cluster_len;
    MPI_Send(&data[0], 1, row_vector, target, 1, MPI_COMM_WORLD);
}

// recv from below (row)
if(can_transfer('d', my_rank, cluster_len, p)){
    int source = my_rank + cluster_len;
    MPI_Status st;
    MPI_Recv(recv_data[1], 1, row_vector, source, 1, MPI_COMM_WORLD, &st);
}

// recv from left
if(can_transfer('l', my_rank, cluster_len, p)){
    int source = my_rank - 1;
    MPI_Status st;
    MPI_Recv(recv_data[2], side_len, MPI_DOUBLE, source, 4, MPI_COMM_WORLD, &st);
}
```

Here we need to make a single MPI_Send() MPI_Recv() call for transfer in a single direction. Note that while receiving a row, we can safely use the row_vector datatype, while we need to use MPI_DOUBLE when recv for a column is made because we want to fit it in a horizontal buffer.

Section 3

Issues faced during testing/development

Following are some mentionable bugs we faced during code development/testing:

1. During MPI_Send()/MPI_Recv() when only one single MPI_DOUBLE was sent/recv, we had a problem in our implemetation where our send/recv buffers were overlapping.
2. During MPI_Pack()/MPI_Unpack(), we were not resetting the value of position variable, thus we were getting segmentation faults
3. During MPI_Unpack() we realized that we were losing the data when using MPI_Isend(), this was because the call was non-blocking and therefore we were upacking before the buffer was filled. This could be fixed in two ways, first using MPI_Wait() for the specific call, and then unpacking. Second, to use a blocking recv so that the unpacking buffer is filled with the sender's data. We used the latter in our implementation (in all of the send/recv calls blocking send/recv are used).

4. While using MPI_Type_vector(), we faced a segfault because we were receiving the col_vector datatype in a horizontal buffer, since the layout is preserved, the recv call was trying to fill the recv buffer out of bounds thus producing the error. We fixed this by using MPI_DOUBLE and receiving one of it at a time.

5. While setting up the NodeAllocator, we faced a package not found error, so we had to manually install psutil . Since at that time, the CSE cluster was not connected to the internet, we had to use this script to manually connect to the internet: `https://github.com/satendrapandeymp/Auth_IITK/blob/master/auth-iitk.py`.

Section 4

Observations

The following 4 plots are corresponding to the number of processes involved, i.e. 16, 36, 49, 64.

- In all of the cases, multiple send/recv takes the highest time. This is due to the overhead for multiple send/recv calls.
- Using MPI_Type_vector takes the smallest time because of minimum overhead of type creation. In pack/unpack, there is an extra overhead due to packing/unpacking, while in send/recv, overhead is introduced due to multiple send/recv calls.

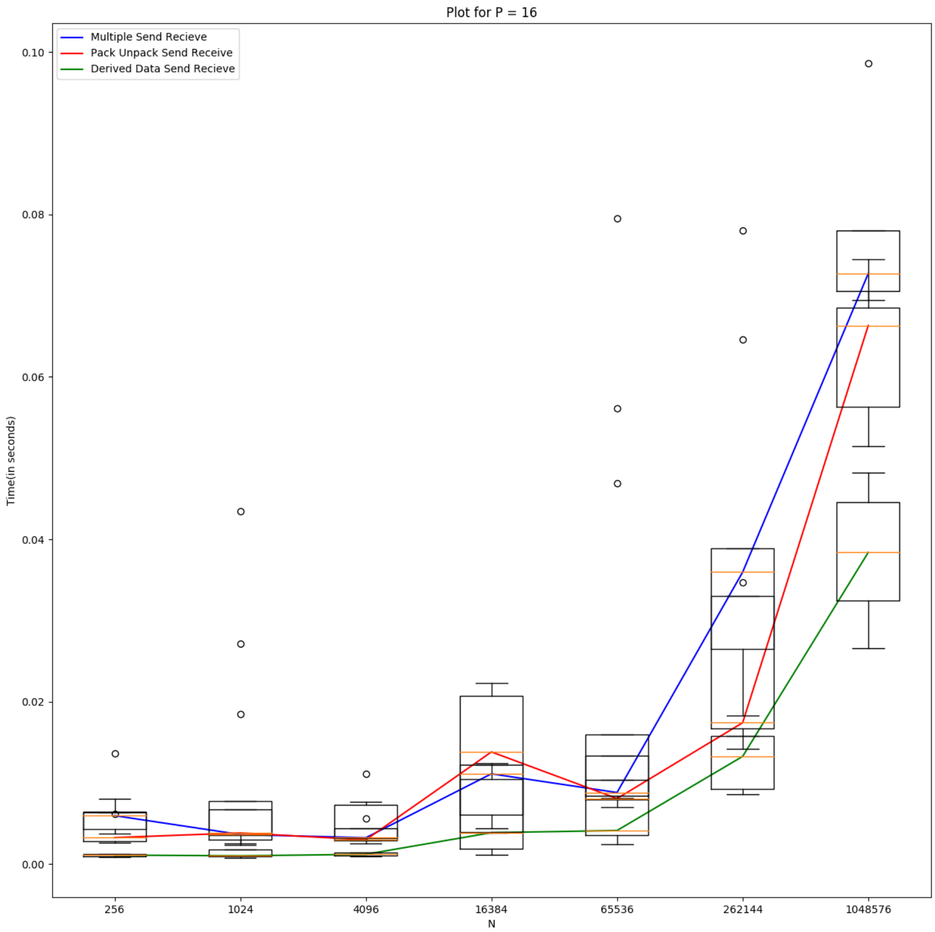

Figure 1: Number of Processes = 16

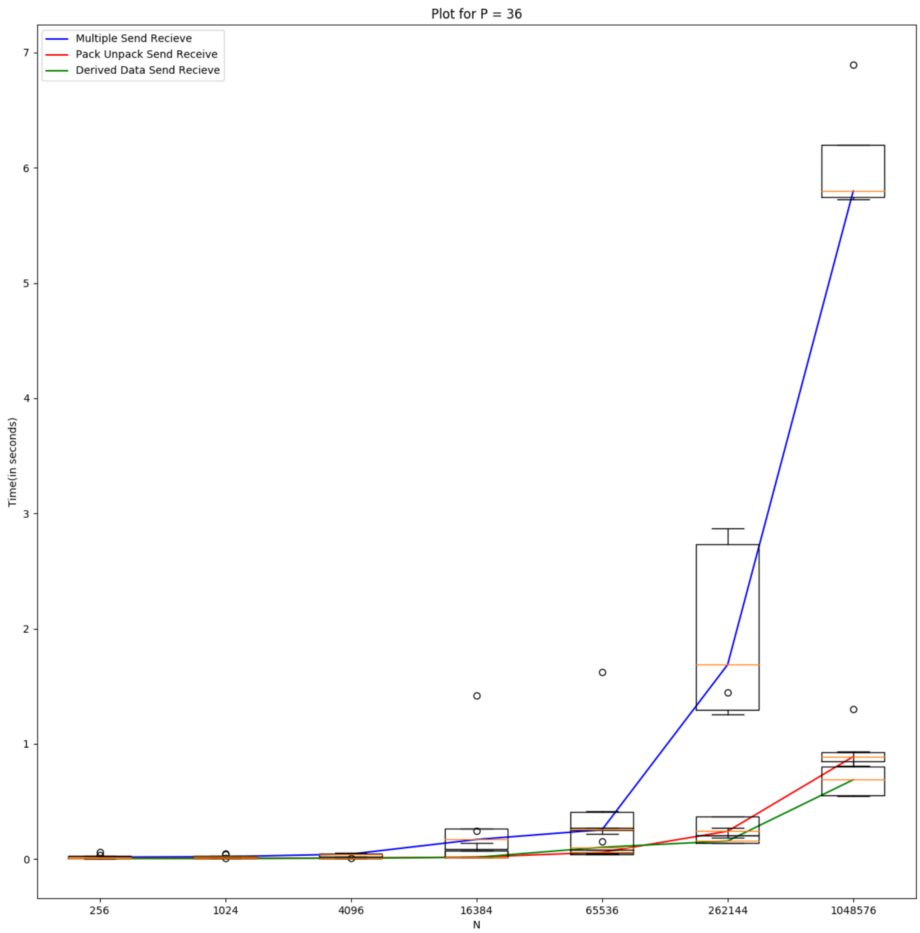

Figure 2: Number of Processes = 36

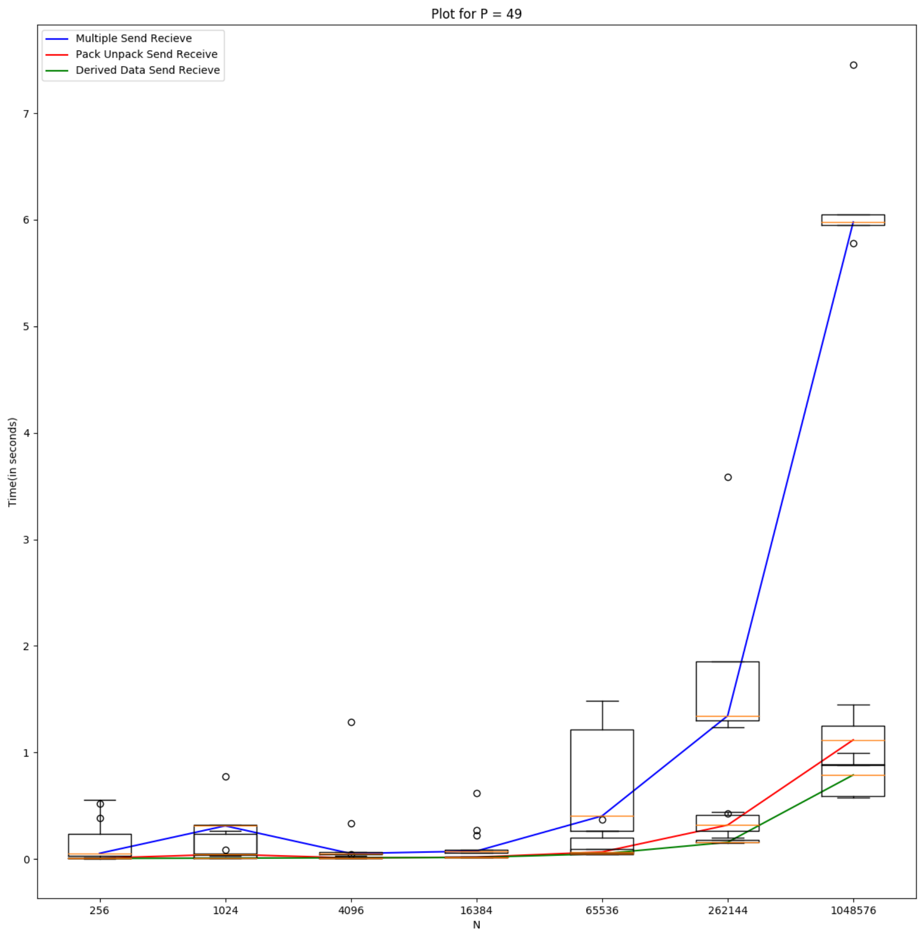

Figure 3: Number of Processes = 49

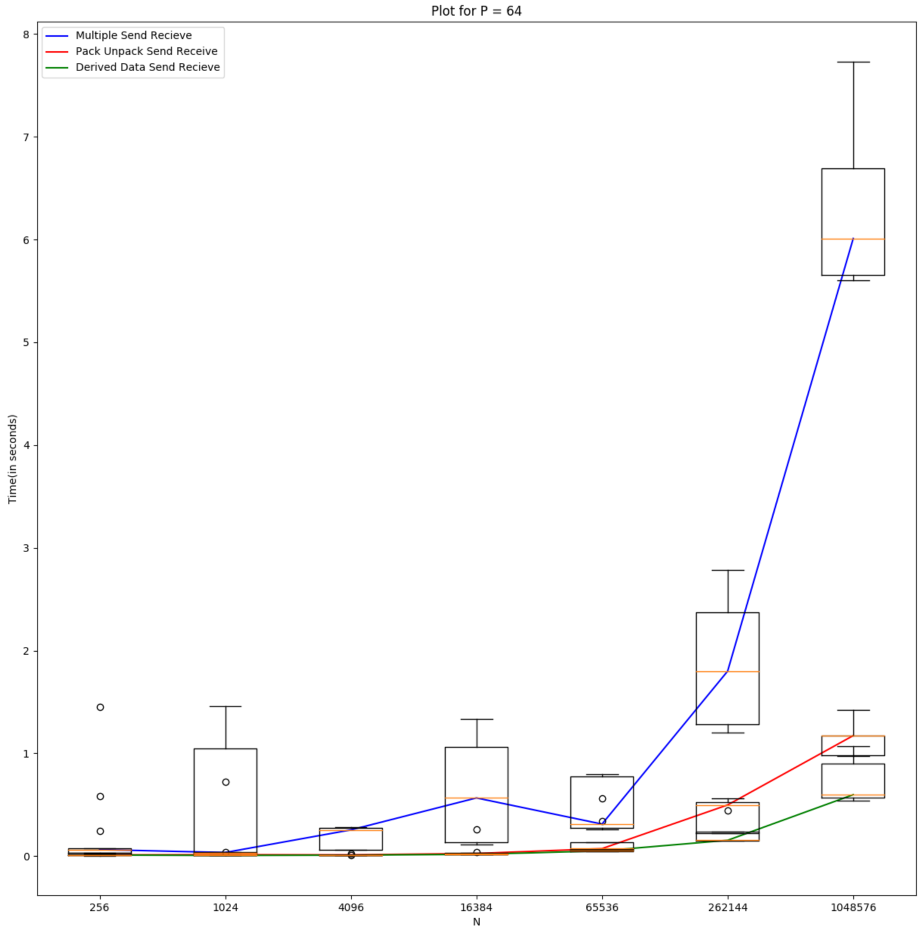

Figure 4: Number of Processes = 64