# Page Rank Algorithm Using MPI

For this problem of assignment, I have implemented a Page Rank algorithm using **four different** approaches.

1. Useing the Iterative Power method to find the eigenvector
2. Using MPI_Allreduce
3. Using MPI_Allgather
4. Using MPI_Irecv and MPI_Isend

**I have also included all four methods codes with this report.**

- **Iterative Power Method To Find Eigen Vector**

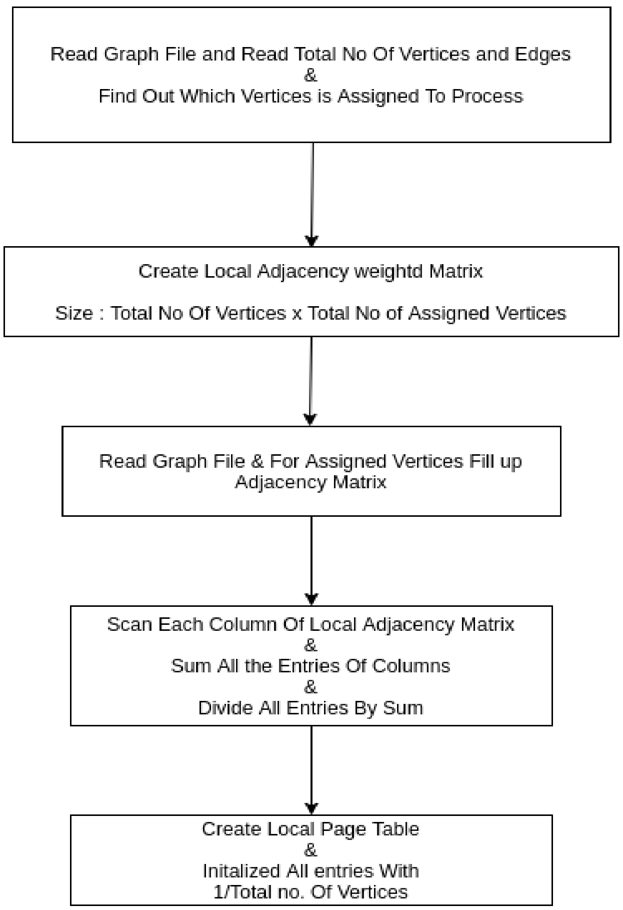
    I have started my implementation with this method, In which we are creating transpose of a weighted adjacent matrix and iteratively we are multiplying a matrix with an initial guess of page rank till convergence.

    This is an easy and straight forward method but after the implementation, I had come to the conclusion that If I will use adjacency matrix then it will be 50000 x 50000 for small graph and **Vector Matrix Multiplication of** (50000 x 50000) and (50000 x 1) will take humongous time.

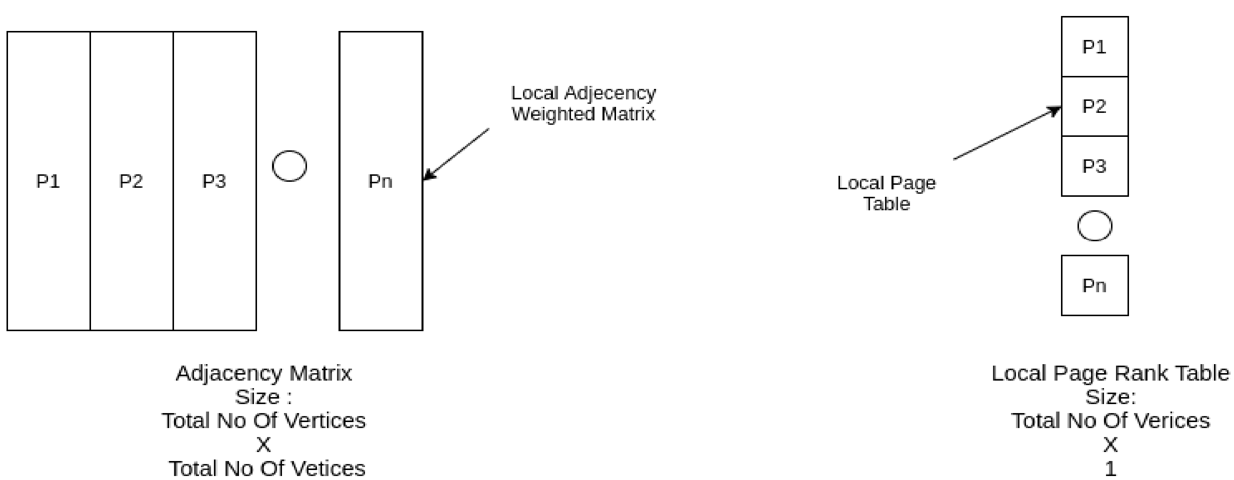
    Even I was able to get output only for 64 processes and it was also taking too much time. In 1 Process even the program was not able to run on the Turing cluster.

**Procedure :**

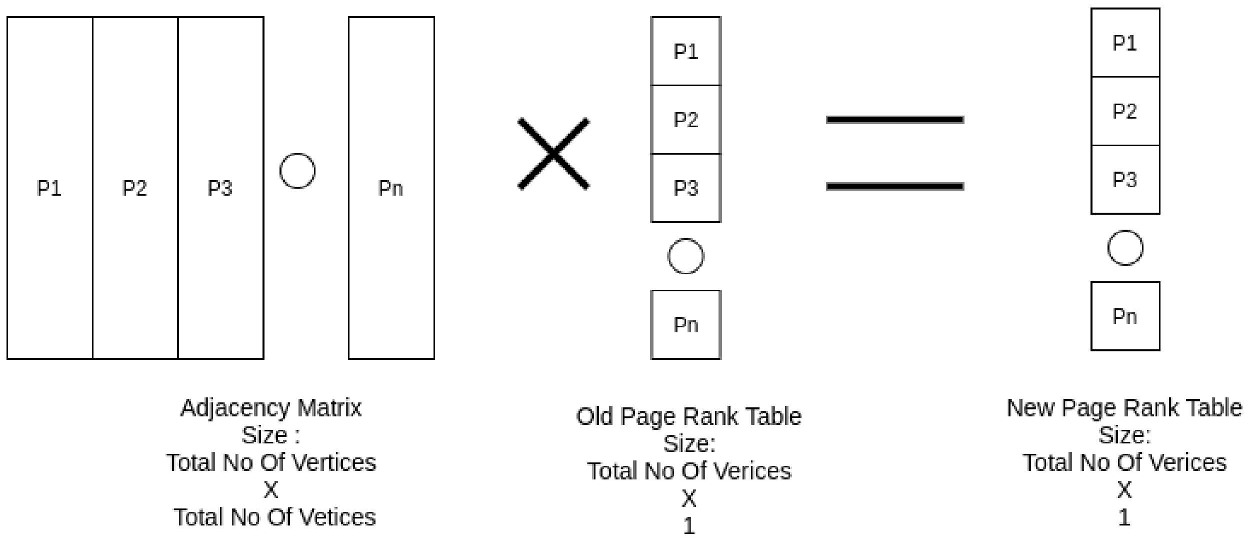
## Procedure Followed By Each Process



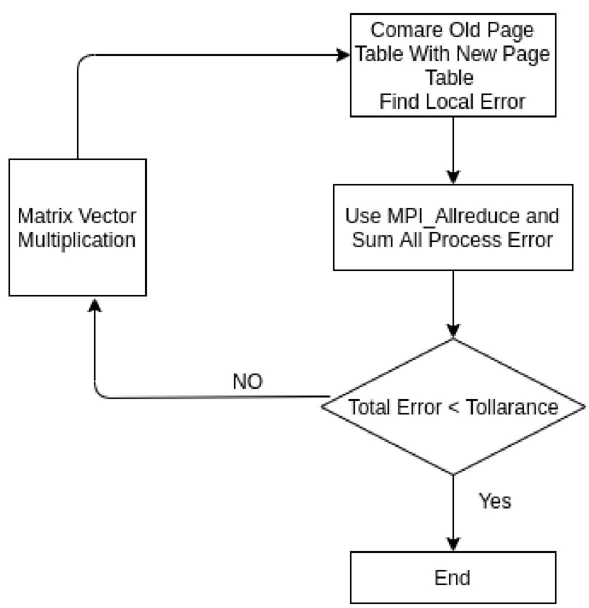

## After The End Of Matrix And Local Page Table Creation Overall Picture



## Matrix Vector Multiplication




## After Each Matrix Multiplication



- **<u>Using AllReduce Method :</u>**

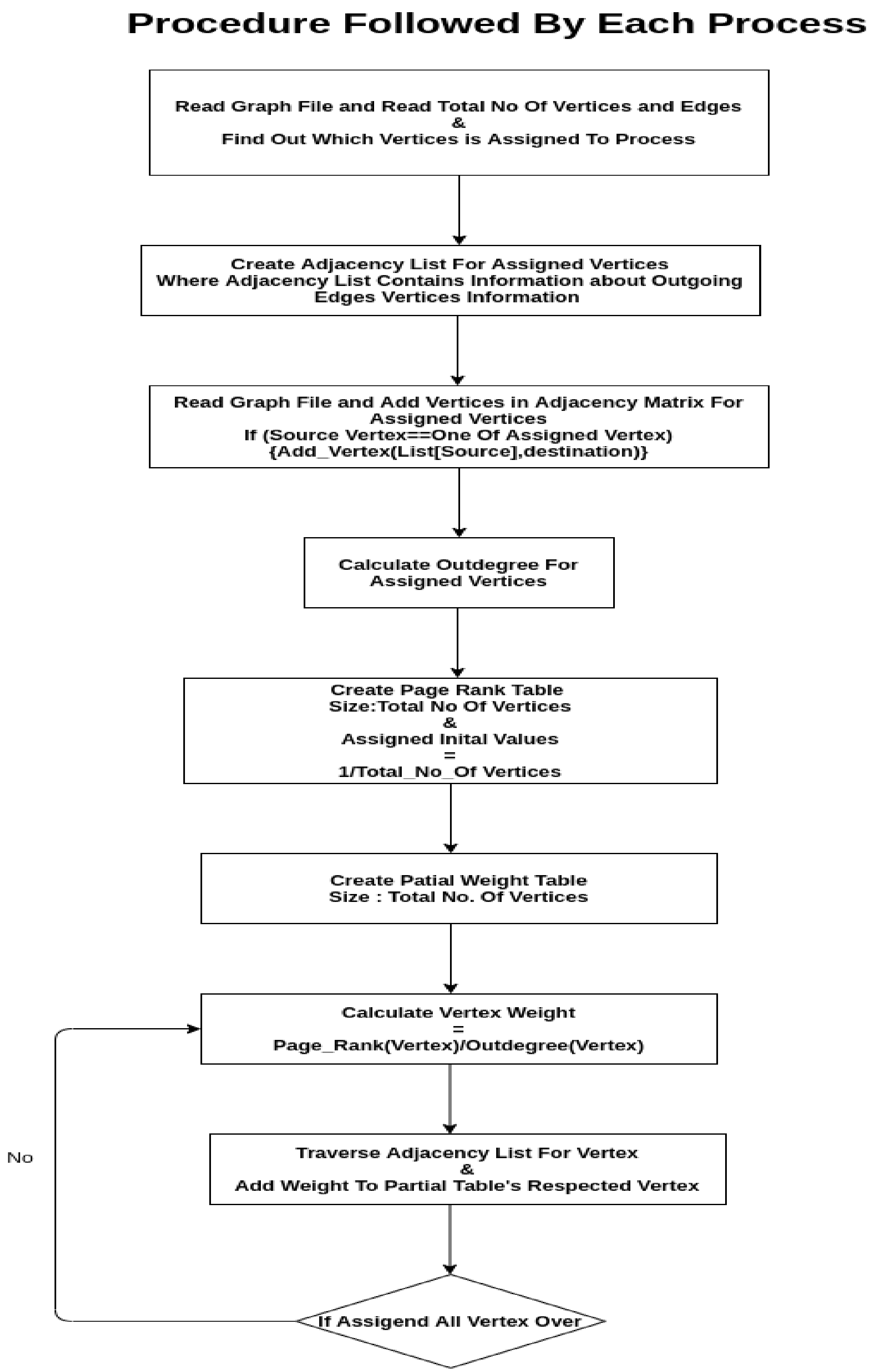
  In this method, I adding the partial weight of each process to get the overall weight for the page rank table.

  Here, I am getting a significant amount of performance improvement than the iterative method but as a no of process increase, my time is also increasing. Even sequential execution time is much lesser than parallel. So, I am getting no speed up here.

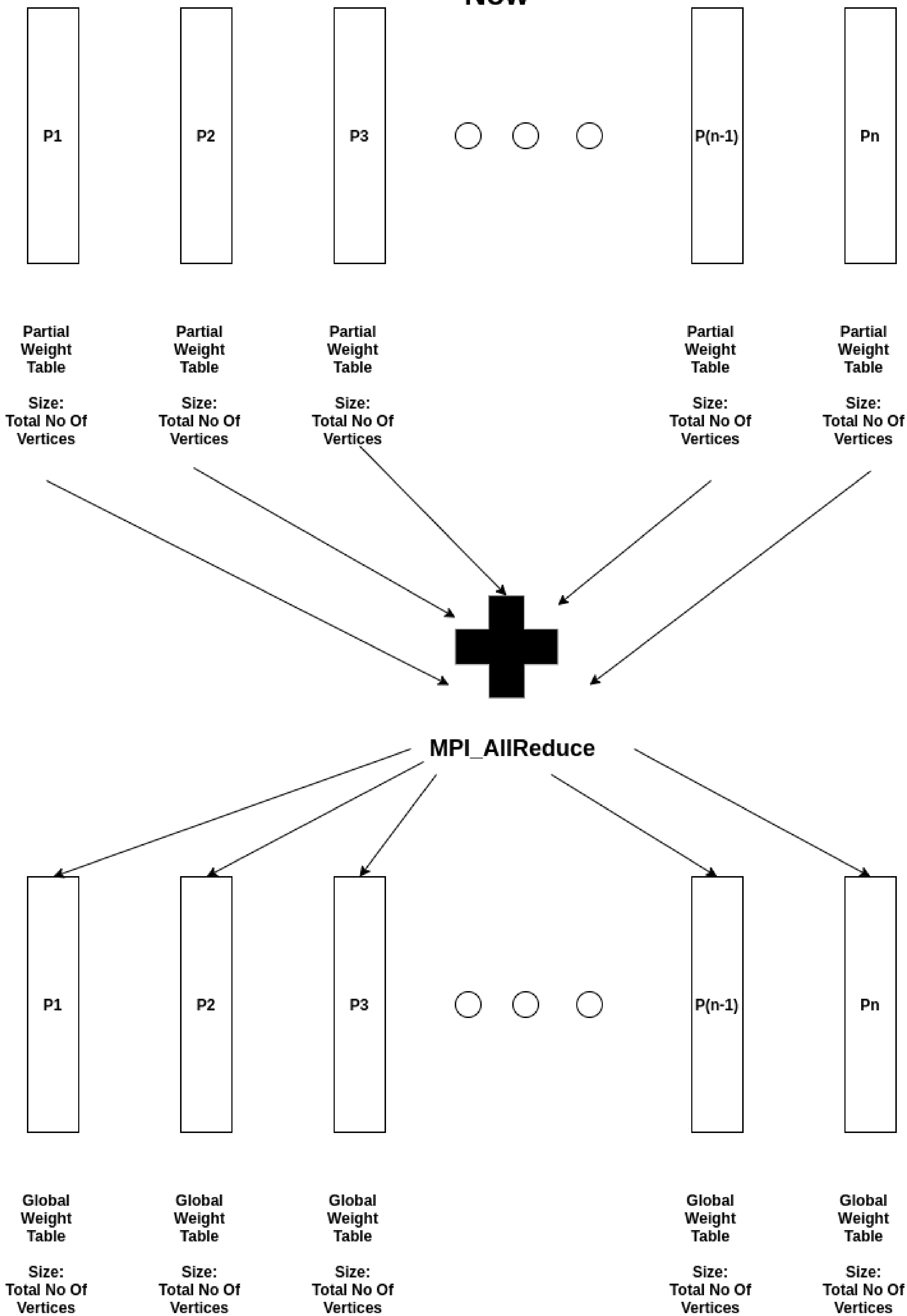
  According to me, Not getting speed up to the sequential is,

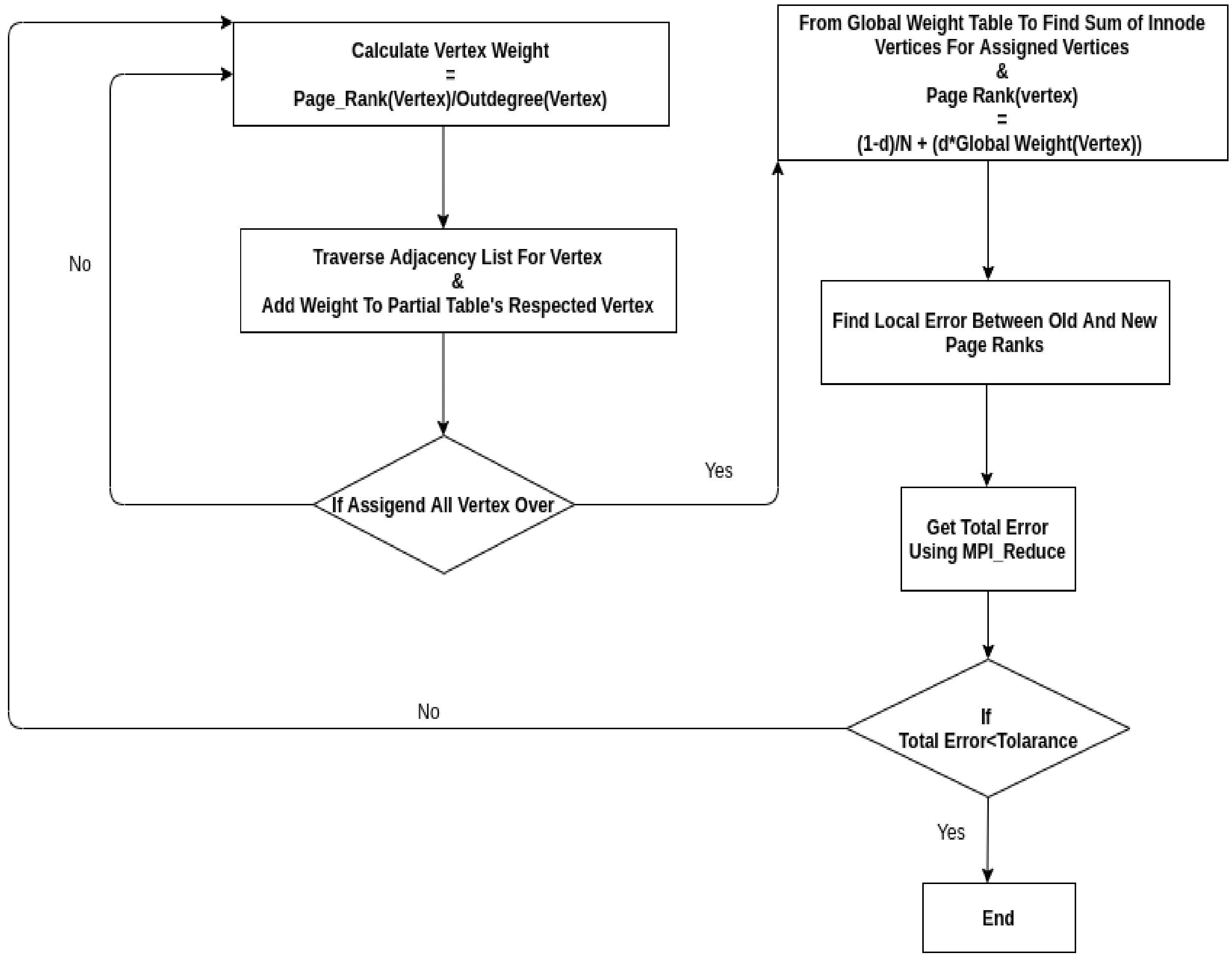
  1. Communication cost is more than computation. Here we are gathering (50000 x number of process) size array over network then do the sum and again broadcast sum so I this communication will take a lot more time than computation which is not with a sequential program
  2. MPI_Allreduce(). We know that MPI_Allreduce internally uses barrier before the beginning of Reduce operation so maybe it is slow down performance.

**Process :**


Procedure Followed By Each Process
Read Graph File and Read Total No Of Vertices and Edges
&
Find Out Which Vertices is Assigned To Process
Create Adjacency List For Assigned Vertices
Where Adjacency List Contains Information about Outgoing
Edges Vertices Information
Read Graph File and Add Vertices in Adjacency Matrix For
Assigned Vertices
If (Source Vertex==One Of Assigned Vertex)
{Add_Vertex(List[Source],destination)}
Calculate Outdegree For
Assigned Vertices
Create Page Rank Table
Size:Total No Of Vertices
&
Assigned Inital Values
=
1/Total_No_Of Vertices
Create Patial Weight Table
Size : Total No. Of Vertices
Calculate Vertex Weight
=
Page_Rank(Vertex)/Outdegree(Vertex)
Traverse Adjacency List For Vertex
&
Add Weight To Partial Table's Respected Vertex
If Assigend All Vertex Over
No

# Overall Picture After Completion Of Till Now






- **Using MPI_AllGather:**

To get speed up, I think that, If I will reduce data that pass over the network then I will get up a speed up. So I come up with a solution and use MPI_AllGather.

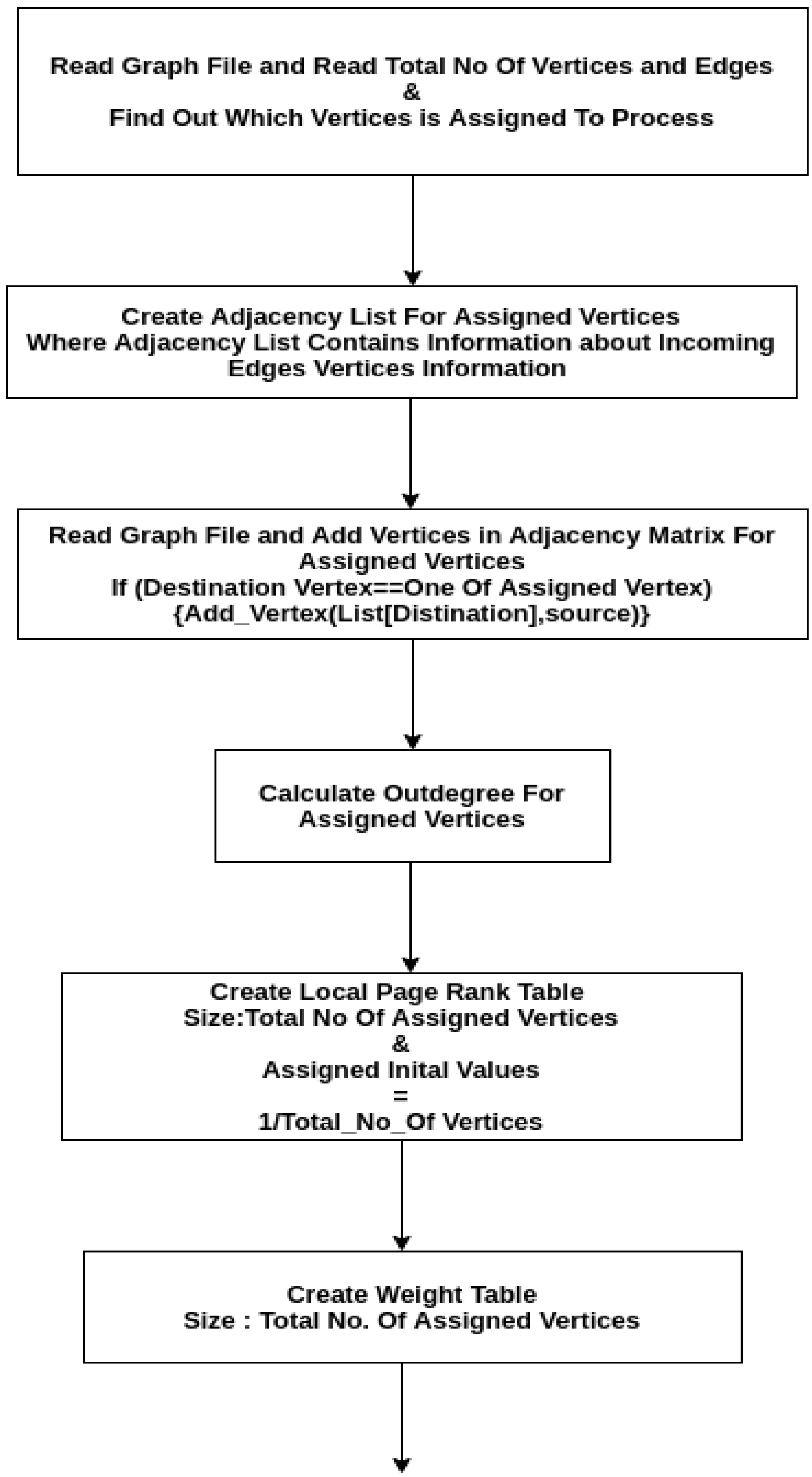
In this approach, I am only sending assigned vertices weight (PR(vertex)/Outdegree(vertex) to other processes and other processes use this weight to calculate a new pagerank.

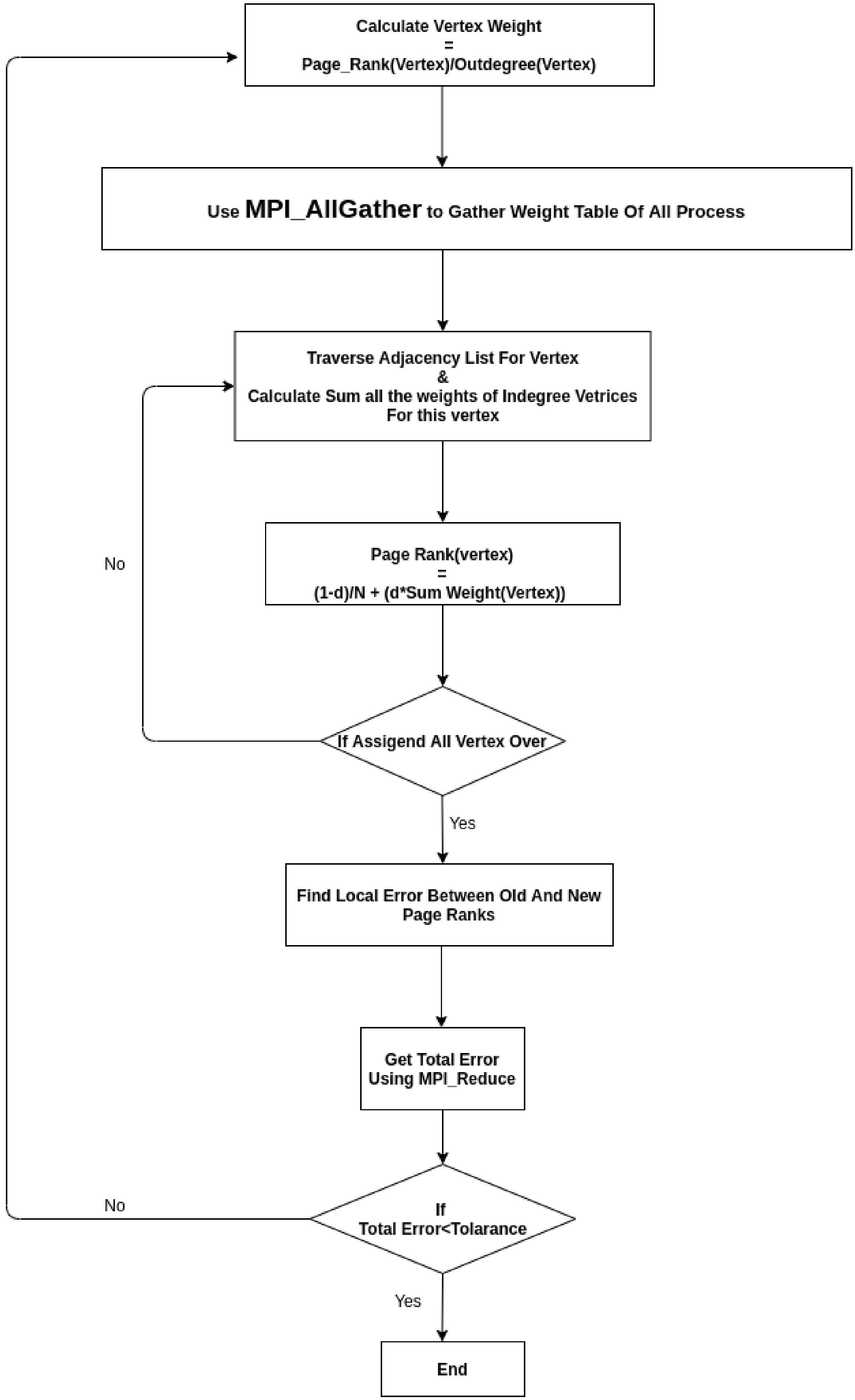
For a small graph, In MPI_Allreduce approach, we are sending (50000 x 1) vector while here We are sending ((50000)/nproc) x 1) vector.

So here I am getting slight improvement than the second approach but still, I am not getting speed up here.

**Process :**

# Procedure Followed By Each Process




Calculate Vertex Weight
=
Page_Rank(Vertex)/Outdegree(Vertex)
Use MPI_AllGather to Gather Weight Table Of All Process
Traverse Adjacency List For Vertex
&
Calculate Sum all the weights of Indegree Vetrices
For this vertex
Page Rank(vertex)
=
(1-d)/N + (d*Sum Weight(Vertex))
No
If Assigend All Vertex Over
Yes
Find Local Error Between Old And New Page Ranks
Get Total Error Using MPI_Reduce
No
If
Total Error<Tolarance
Yes
End

- **Using MPI_ISend And MPI_IRecev :**

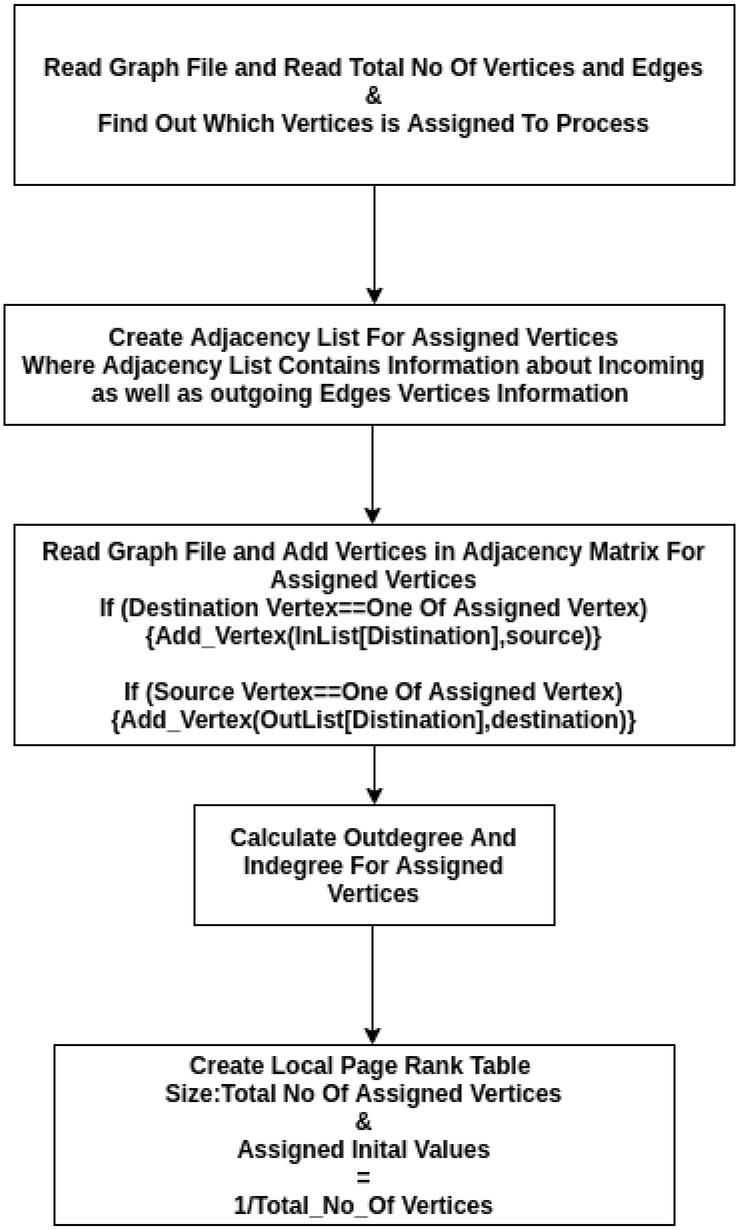
  In all the previous two approaches, we are sending data to the other process whether it's required or not, and also with a blocking communication call.

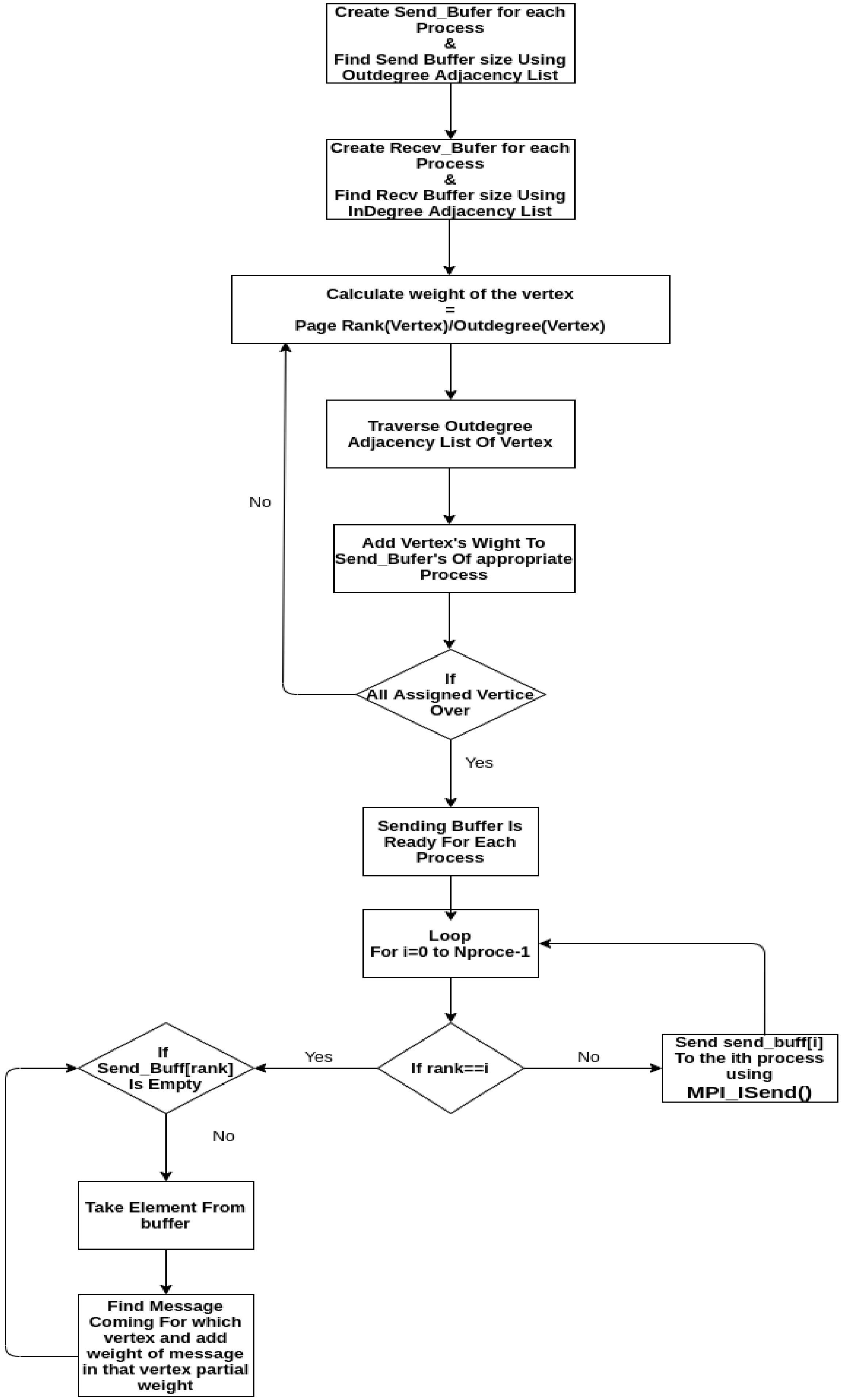
  So, I get an idea that to just send weights to the process which requires only i.e communicate with the only boundary process. This will reduce lots of communication overhead and we can send this data using MPI_ISend and MPI_Irecev. Which will also overlap communication with computation.

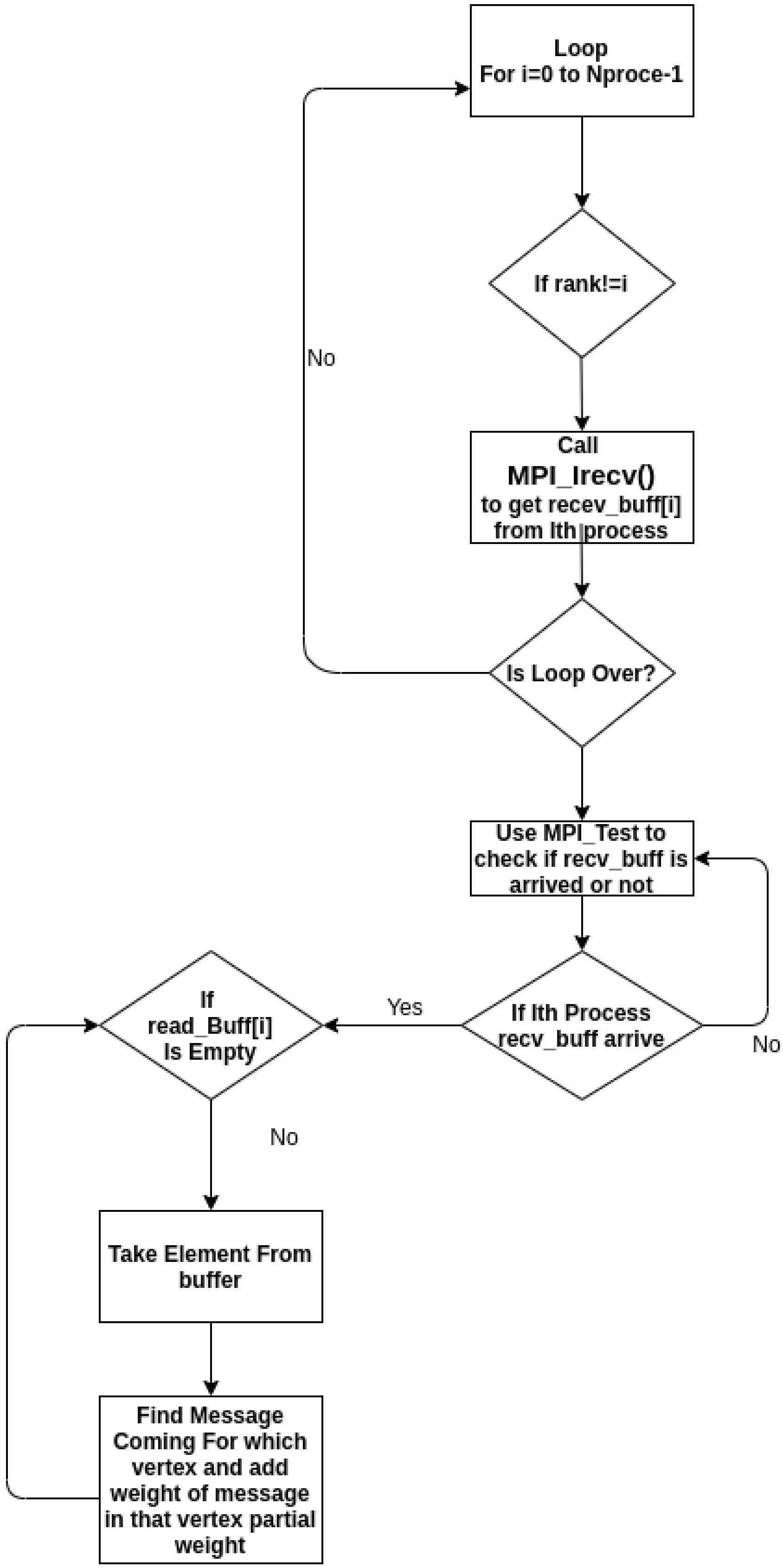
  Using this approach I am getting a significant amount of performance improvement and speed up for the large graph.

**Process :**

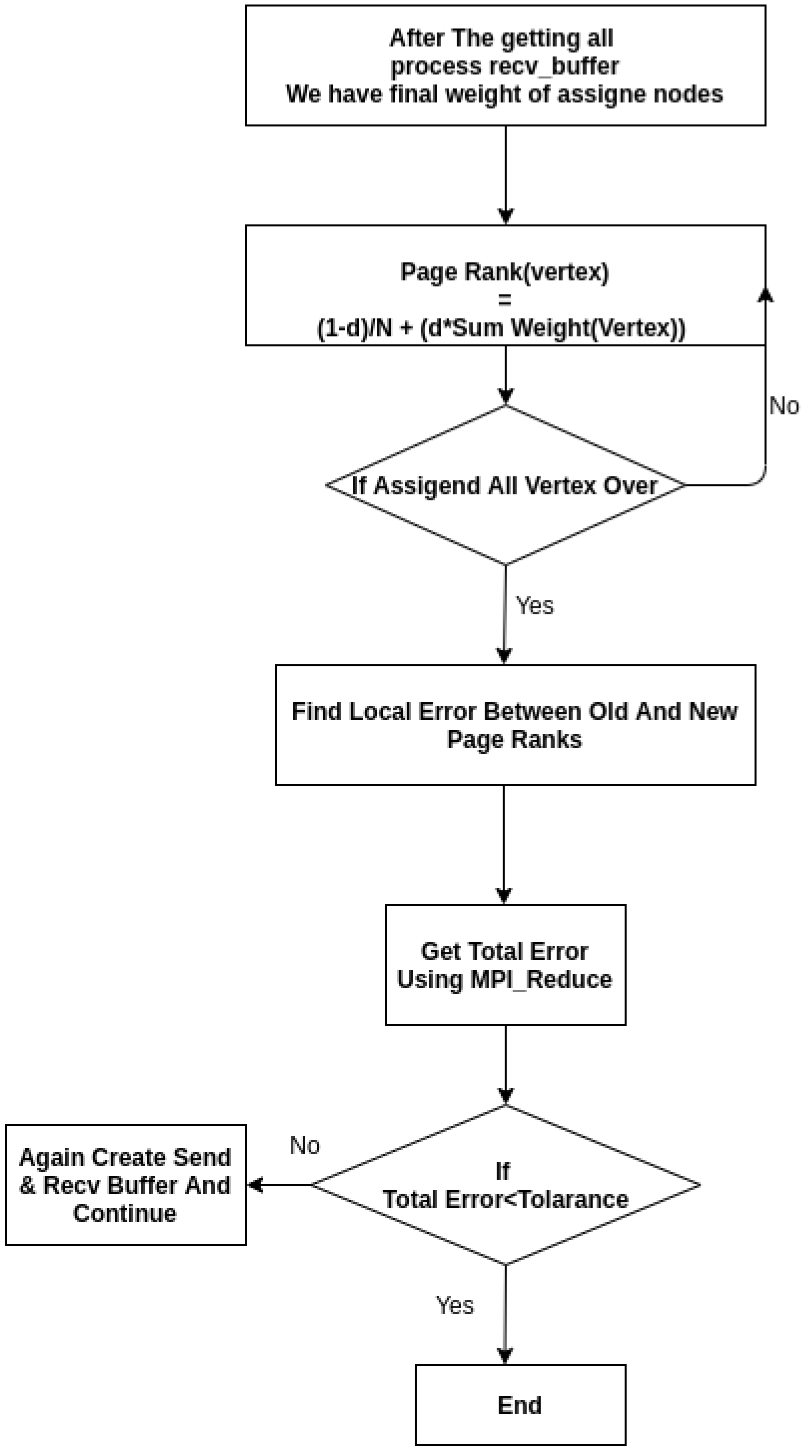
# Procedure Followed By Each Process




Create Send_Bufer for each Process & Find Send Buffer size Using Outdegree Adjacency List
Create Recev_Bufer for each Process & Find Recv Buffer size Using InDegree Adjacency List
Calculate weight of the vertex = Page Rank(Vertex)/Outdegree(Vertex)
Traverse Outdegree Adjacency List Of Vertex
Add Vertex's Wight To Send_Bufer's Of appropriate Process
If All Assigned Vertice Over
No
Yes
Sending Buffer Is Ready For Each Process
Loop For i=0 to Nproce-1
If rank==i
Yes
No
Send send_buff[i] To the ith process using MPI_ISend()
If Send_Buff[rank] Is Empty
No
Take Element From buffer
Find Message Coming For which vertex and add weight of message in that vertex partial weight



Loop
For i=0 to Nproce-1
If rank!=i
Call
MPI_Irecv()
to get recev_buff[i]
from Ith process
Is Loop Over?
No
Use MPI_Test to check if recv_buff is arrived or not
If Ith Process recv_buff arrive
No
Yes
If read_Buff[i] Is Empty
No
Take Element From buffer
Find Message Coming For which vertex and add weight of message in that vertex partial weight

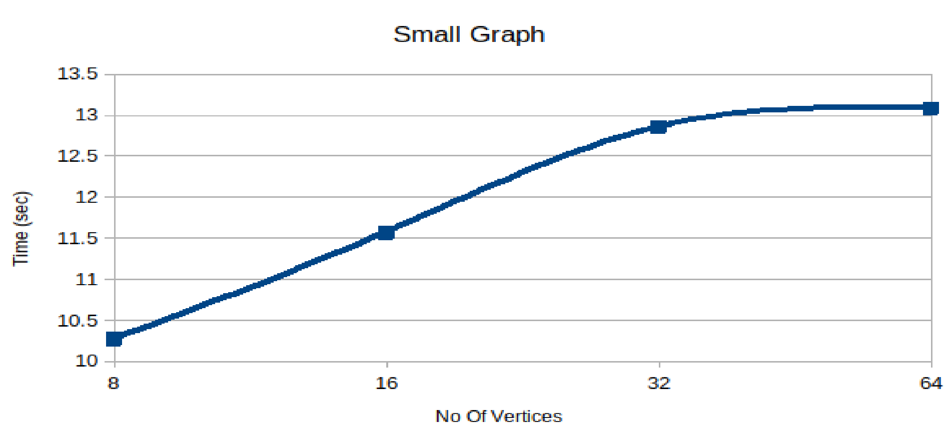

After The getting all
process recv_buffer
We have final weight of assigne nodes
Page Rank(vertex)
=
(1-d)/N + (d*Sum Weight(Vertex))
No
If Assigend All Vertex Over
Yes
Find Local Error Between Old And New Page Ranks
Get Total Error Using MPI_Reduce
No
Again Create Send & Recv Buffer And Continue
If
Total Error<Tolarance
Yes
End

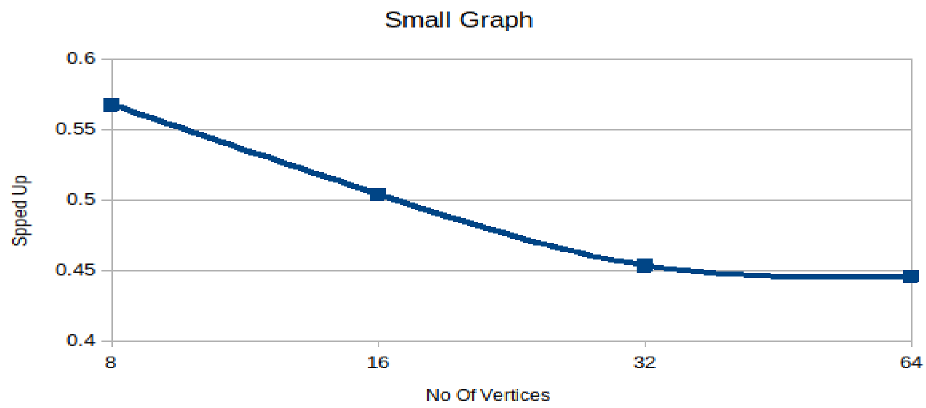
## Performance Of The Last Approach :

## Small Graph :

| Sequential Execution Time | |
|---|---|
| **Execution Number** | **Time (Sec)** |
| 1 | 5.86435 |
| 2 | 5.89143 |
| 3 | 5.87541 |
| 4 | 5.76951 |
| 5 | 5.73856 |
| **Average Execution Time** | **5.827852** |

| No Of Processes | 1 | 2 | 3 | 4 | 5 | Average Time(sec) | Speed Up |
|---|---|---|---|---|---|---|---|
| **8** | 10.2349 | 9.87571 | 10.382 | 10.3375 | 10.519 | **10.269822** | **0.567473516** |
| **16** | 10.9693 | 11.6272 | 11.9466 | 11.6945 | 11.6042 | **11.56836** | **0.50377512** |
| **32** | 12.2061 | 13.388 | 13.0413 | 13.451 | 12.1792 | **12.85312** | **0.45341924** |
| **64** | 12.8356 | 12.6631 | 13.3358 | 13.5801 | 12.9821 | **13.07934** | **0.44557691** |

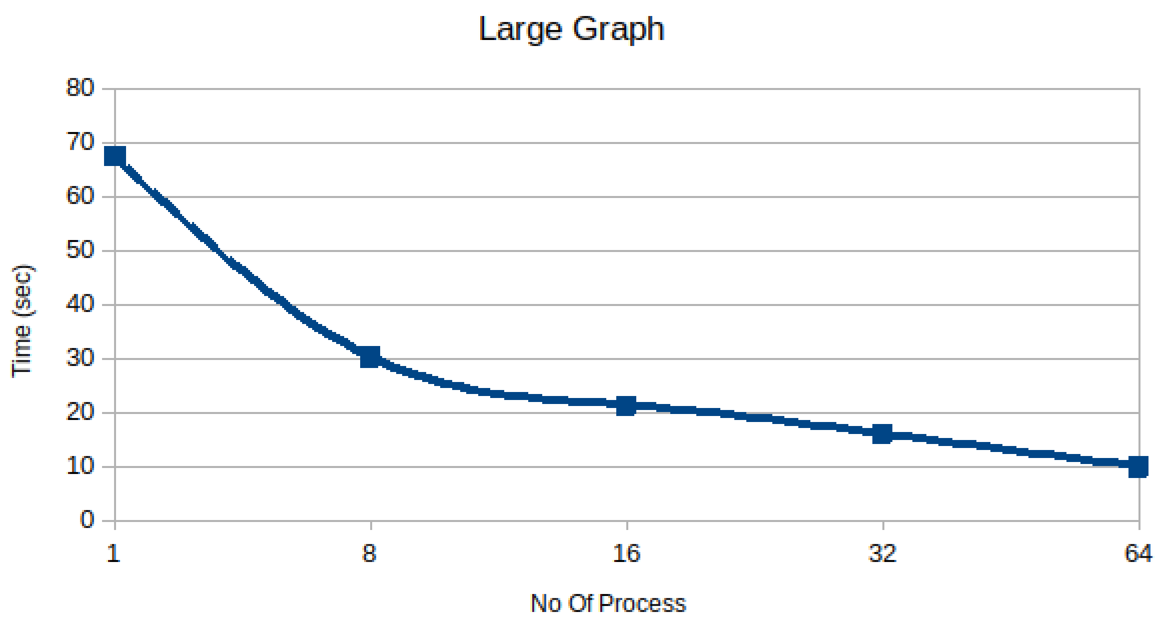


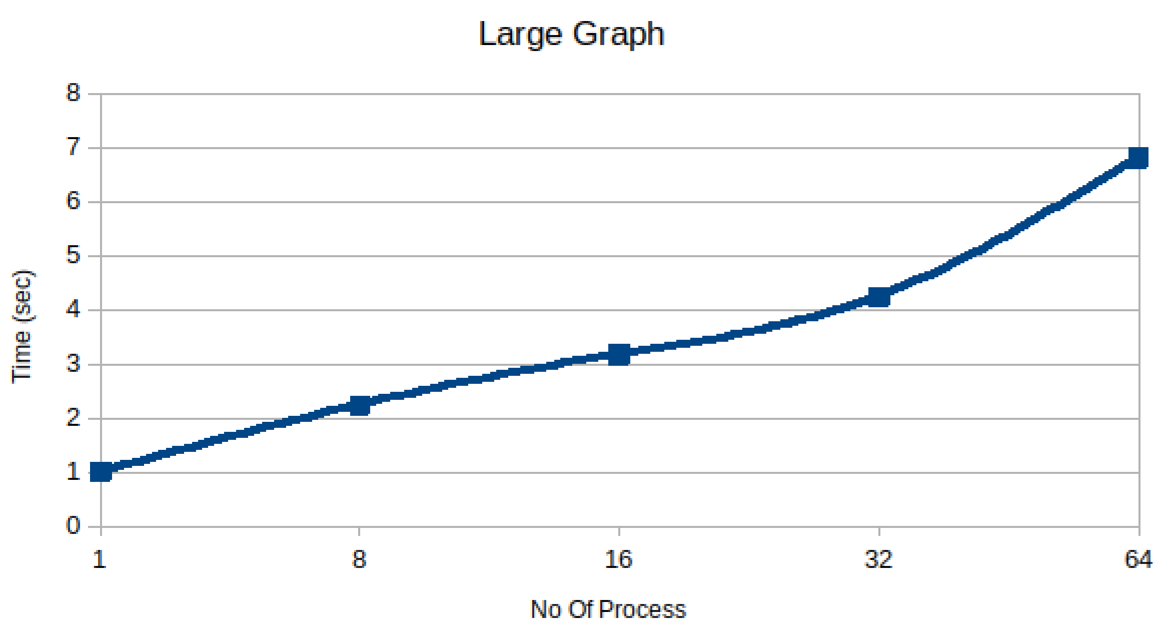



## Large Graph

| Sequential Execution Time | |
|---|---|
| **Execution Number** | **Time (Sec)** |
| 1 | 64.8979 |
| 2 | 74.6496 |
| 3 | 60.5902 |
| 4 | 63.337 |
| 5 | 73.7916 |
| **Average Execution Time** | **67.45326** |

| No Of Processes | 1 | 2 | 3 | 4 | 5 | Average Time(sec) | Speed Up |
|---|---|---|---|---|---|---|---|
| **8** | 30.2417 | 29.9857 | 30.2145 | 30.3854 | 30.1478 | **30.19502** | **2.2339200** |
| **16** | 20.4125 | 21.5214 | 20.9821 | 21.6752 | 21.6542 | **21.249086** | **3.1744075** |
| **32** | 16.239 | 16.0176 | 15.9898 | 15.2555 | 16.1792 | **15.93622** | **4.2327013** |
| **64** | 9.59988 | 9.55722 | 10.1815 | 10.288 | 9.8965 | **9.90462** | **6.8102824** |




**Remarks :**

We can still get better performance if we will send only weight which has changed in the previous iteration.