Introduction:

In this paper we explore Gossip based protocols to compute various aggregation functions. Gossip computation is used in distributed networks inorder to pass messages between the various nodes. It is particularly useful for aggregation because it helps calculate the functions for a large and dynamic network, with different nodes having different values. In this paper we implement Gossip in Erlang to compute the average, minimum, maximum and median values of a file which is divided into fragments, with each fragment stored at one or more node. We also implement rumour spreading vi gossip, which disseminates information into the entire network. An update originating at a single node is given to all the nodes in the network. Also, any node can retrieve information about a particular fragment by communicating with the nodes.

In this paper, we take you through the general framework first. The section on Implementation is divided into two major parts, General Gossip Functions and the Problem Statement. While the former explains the methods and setup of the whole gossip, the latter is further divided into five different parts, each explaining in detail the methodology behind each problem statement. An Algorithm/Method for that function is followed by the results and conclusions we can draw from that part. Any Assumptions made are also stated. Future works have been included at the end of the paper.

Implementation:

1) Gossip Framework-
We have used gossip framework for solving all problems. We do getneighbours, selectneighbours, spawn thread, starting times and managing messages and fragment tasks in this framework; these tasks are generic and useful for all 5 problems. Following is the flow of tasks that happen in this framework.

1) We generate topology and create fragments. Their creation is explained in section 2 and section 3.
2) Spawn nodes and initialize them fragment, message and other problem specific variables. These variables change according to problem and will be explained in initialization routine. We store the process Ids and pass then as message to each process initially. Although process has all pids it only uses pids of its neighbors which is local information available at every node.
3) Start the timer of each node and send message to other nodes after every tick of the timer. We do not assume that timers are synchronized. Since we are sending message after every tick, time taken is proportional to no of messages sent.
4) For sending message, we use probability transition matrix to select neighbor. We divide number from 0 to 1 according to probabilities and generate random number and check range of probability that number is in. Thus if probability of node is higher than random number will fall in range of that node more often and node will be get selected usually.
5) When node receives message it processes it according to problem it is solving and replies back to sender. Replying back to sending node is necessary for gossip to converge and so we are performing it for every task.

Assumptions
1) We assume that nodes don’t fail. But the algorithms that we propose are robust and can work under conditions of failure as well except for retrieve. Retrieve nodes must not fail till execution of a retrieve message. They can fail in case of before or after starting gossip for retrieve message.
2) The probability transition matrix is irreducible. Thus every node is connected to some node and we don’t get disconnected topology.
3) The probability transition matrix is aperiodic. We are achieving aperiodicity by making having some probability that matrix talks to itself or talks to none of its neighbors. This is necessary so that average values do not oscillate between two extreme values.
4) The probability transition matrix is doubly stochastic, so that all probabilities of node talking to neighbor add up to zero.
5) We make sure that preservation of mass for nodes while averaging is achieved. This is done by making every node reply to sending node with its message.

**Gossiping Accross 2 Virtual Machines:**

Using Erlang, we can connect two Machines in order to pass information between them. We set the cookies on each system with the same value. While spawning the threads, include the Machine-Name as a parameter in spawn/4. Please refer to Appendix for the Screenshots displaying communication between 2 virtual machines

**2) Fragment Generation**

Use python script to generate a large number of random floating point numbers within a specific range. Write it in a file (randnofile). Compute max, min, median and average on these numbers. Use these results to verify the correctness of computations using Gossip method.

**Algorithm**

1) Read the ‘randnofile’ and create a list of numbers and index each value. We are indexing each value for sake of simplicity and we can also index list of values.
2) Create empty fragments equal to number of nodes.
3) Select one index value pair and insert it in replication factor number of fragments. The fragments where these values are inserted are generated randomly.
4) Perform above action till all index value pairs in ‘randnofile’ are placed in some fragment.

The final generated list of fragments is thus unequal in size and has all the values replicated across fragments a fixed number of times. We call this as its replication factor of file. Since we are generating random numbers uniformly, the distribution of size of fragment is cluster around some value.

**3) Topology Generation**

Generate Topology using Python Script. The number of Nodes and Neighbor Radius limit is passed as parameters. Generates a topology with N nodes with the number of neighbors determined by neighborhood threshold. N number of random coordinates are generated in the coordinate space [0,1). For determining neighbors a coordinate is chosen from the list of coordinates and the distance between that coordinate and rest of the coordinates is calculated. If the distance falls under a value as specified by Radius, it is added as a neighbour otherwise it is ignored. After the above step each node has a list of neighbours associated with it.

To calculate transition probabilities for the markov chain, the method given by Dr. Kalpakis based on degree of nodes was used. Also it was made sure that Probability Transition Matrix is not lazy by averaging it with Identity matrix.

**Algorithm**

Function Topology (N = Number of Nodes, Radius = Neighbourhood Radius Limit)
1. Generate N random coordinates in [0,1)
2. Select a coordinate and calculate it's distance to all other coordinates
3. If the distance is under Radius, add the other node to it's list of neighbours
4. Repeat step 2 and 3 for all the set of coordinates
5. For each node in list of nodes(coordinates), repeat steps 6, 7, 8, 9
6. a node is selected from it's list of neighbours.
7. P(i,j) = 1/\[1 + \text{max}(d(i), d(j))\] i, j = Node, it's neighbour chosen in above step.
8. P(i,i) = 1 - sum_{k in N(i)} P(i,k).
9. The probabilities are averaged with Identity matrix and stored as a tuple associated with a Node.
10. Write the probabilities associated with a node in a file
11. Write the number of neighbours with a node in another file.
Problem Statements:

1) **Minimum/ Maximum**: Compute the minimum and maximum value in F and store them at node

**Method:**

1. **Initialization**
   The process nodes are spawned and they compute maximum/minimum of their fragments and it is passed around as message while Gossiping.

2. **Gossip Between Nodes**
   Once the framework has been initialized as explained earlier and all the process nodes have spawned, the computation is performed using gossip as follows.
   1. Each node selects one of it’s neighbours from the neighbours list and message that node the message with value corresponding to computation, it’s Process id and Function.
   2. The other node upon receiving message finds the maximum or minimum as specified in Function of it’s own message. It updates the message based on the earlier computation and sends a reply to sender node with the updated message.
   3. The sender node upon receipt of reply updates the message and receives a message from it’s neighbours or sends a message to one of it’s neighbours.

**Results/Graphs:**

![Graph 1](image1.png)

**Maximum**

| NT=| RF=1, Acc=0.01 |
|---|---|---|
| 0.2 | ![Graph 2](image2.png) |
| 0.3 |

NT=Neighborhood Threshold, RF= Replication Factor, Acc=Acceptable Error (%), Convergence = No. of Msgs to reach convergence with defined parameters.

![Graph 3](image3.png)

**Minimum**

| NT=| RF=1, Acc=0.01 |
|---|---|---|
| 0.2 | ![Graph 4](image4.png) |
| 0.3 |

NT=Neighborhood Threshold, RF= Replication Factor, Acc=Acceptable Error(%) , No. of Msgs to reach convergence with defined parameters.
From the above graphs for Minimum/Maximum, we can see that changing various parameters of the gossip function exponentially changes the no.of messages required to converge. We changed the number of nodes from 100 to 1500 and our gossip was successful for all. This algorithm is scalable for very large number of nodes as well. However, due to computation limitations, we couldn't get an output for them.

**Conclusion:**
Maximum and minimum computation was completed successfully for upto 1500 nodes with 20,000 elements distributed in fragments of these nodes with various replication factors. The accurate maximum and minimum among all the elements were found out using Gossip algorithm and the results were verified. The following results could be concluded from the experiments conducted:
1. With increasing number of nodes, the time taken for the computation to converge on the maximum element was found to be increasing which was the expected behavior.
2. Time taken for convergence was inversely proportional to the number of neighbours which was changed in topology generating program by varying the neighbourhood radius. This was expected as higher number of neighbours increases the conductance of the graph.
3. Increasing the replication factor also had a positive impact on the convergence time. This is due to the fact that as numbers are present over more number of nodes, it computes the maximum and minimum value faster which in turns becomes the message that is gossiped around.

2) **Average:** Compute the average of the values in F and store it at all the nodes.

**Method:**
1. **Initialization**
The process nodes are spawned and they compute sum of elements, length of fragments and it is initialized as the message to be gossiped around.
2. **Gossip Between Nodes:**
Once the framework has been initialized as explained earlier and all the process nodes have spawned, the computation is performed using gossip as follows:
1. Each node selects one of it's neighbours from the neighbours list and message that node the message with details value of corresponding computation, it’s Process id and Function.
2. The other node upon receiving message averages the sum, length in message it received with it’s own length, sum of fragments. It updates the message based on the earlier computation and sends a reply to sender node with the updated message.
3. The sender node upon receipt of reply updates the message and receives a message from it’s neighbours or sends a message to one of it’s neighbours.

**Results/Graphs:**

![Average Graphs](image)

NT=Neighborhood Threshold, RF= Replication Factor, Acc=Acceptable Error(%), No. of Msgs to reach convergence with defined params
Conclusions
For determining convergence, the accuracy required in the final result had to be relaxed compared to the earlier computation of maximum and minimum. From results shown, we derived following conclusions:
1) As the replication factor increases, the number of fragments also increases, thus as it should be logically, the graph shows that the no. of messages required to converge to a common value decreases.
2) As the acceptable error in our computation is relaxed, the no. of messages required for the computation decreases.
3) As the number of nodes are increased, the time required for convergence increases.

3) Median: To Compute the Median of the given values of a file.

Method:
We have solved problem of finding median in distributed Environment. We solve this problem using only local information. This approach works with fixed replication factor and when all data is distributed across network. Time required has upper bound O(N* log n). Then we provide some advice on how to make this implementation of algorithm fault tolerant.

Algorithm:
1) Nodes are created and every node gets fragment containing random values and each value in fragment is indexed. Our framework can also supports indexing of list of values, but we have not tested it for sake of simplicity.
2) Computation of median is started by first node, so we are initializing first node's message with its local median and with number of values less than median which will be half the size of fragment of first node. All other nodes are will be initialized with zero.
3) Every node requires following data structures:
   1) List of its neighbours Identity for communication.
   2) Process ids of all nodes : Although we are storing Pids for all neighbours, we are using only process ids of neighbours. The Entire list is for sake of simplicity and debugging.
   3) Message which is to be passed to other nodes during gossip.
   4) Total no of Elements in fragment : This can be easily calculated by averaging over gossip where every node gossip number of elements it has and we calculate average by the end of convergence using above method. Since we can also calculate number of nodes using averaging over gossip where only one node has value 1 and all other node has value 0. And if we take reciprocal of it we get number of nodes. This increases number of gossip rounds by 2. Here we are just assuming that every node knows this information but this assumption does not affect validity of algorithm.
   5) Minindex and Maxindex (Minmaxelement) : This is tuple of two indexes which points to range of values median can take in sorted fragment.
   6) CurrentPid : This is process id of thread which is starting the gossip algorithm.
   7) Replication Factor : Our algorithm works with replication of fragments as well. But Replication factor for each fragments must be same.
   8) Iterations : This is used to count number of events occurring at each node. After certain number of events we are just stopping gossip.

4) The messaging in gossip framework happens same way as explained in earlier section. But the way in which nodes handle receiving of messages is different. When node receives a value of first node for first time, it calculates number of values in fragment that are less than value first node gossiped. When it receives message next time it averages own value and value it got through gossip. This process is similar to averaging process explained in !!averaging!!

5) When network converges we will get average number of elements less than the value first node gossiped. We are concluding network has converged after sending some fixed number of messages. This number of messages can be calculated by every node using gossip.

6) If number of elements less than the value gossiped by first node is less than half of total number of values in all nodes, then we will update Minindex to number of elements less than gossiped value, otherwise we will update Maxindex. If number of elements less than the value gossiped by first node is equal to half of total number of values in all nodes, then value gossiped by first node is median.

7) The next value chosen for gossip by first nodes is local median of values between Minindex and Maxindex in sorted fragment. There can be better heuristics than this but in case of uniformly distributed random numbers it reduces possibility
of median set by almost half. So, this is heuristic we are using.
8) Once all values for average are checked by first node, it gives this responsibility to other node. If it has possible median it does gossip and checks it, otherwise it forwards responsibility to one of it neighbours.
9) This approach is similar to token based approach where node having token will checks its values for median, but we are performing random walk on token. Thus, after certain time, all nodes would have had chance to check their value for median. There are better approaches than this which might take lesser time like selecting node which has maximum number of elements which can be median, but this approach is simpler to implement.

Analysis:
This Algorithm divides the set of all values into two parts: one in which we can contain median and one which cannot. In this the time required is of order of \( O(\text{time to check whether every elements is median}) \) which can be given as
\[
\text{Time Complexity} = O(\text{Number of values} \times \text{time for convergence})
\]
Time required for convergence is of order of \( \log n \) where \( n \) is no of nodes. Thus, Time Complexity = \( O(N \times \log n) \) where \( N \) is number of values in all fragments. This Complexity is upper bound and the Complexity of finding median will always be less than this value.

Node Failure:
Our algorithm can tolerate failure of any node other than node having token. Node can voluntarily go away after sending token to any other node. We used token based algorithm because it was simpler to implement and it is just used to select the node which will start gossip. This can be any node which has potential median depending on Minmaxelement values. So, There are other ways like selecting a node which has maximum number of elements which can be known through gossip. This will require extra round of gossip, but time to find median will be reduced. Also calculation of median will become robust. These changes can be easily implemented in our framework just by changing selectneighbour function.

4) Update: Update the contents of fragment i at each node that may have a copy of it. The update originates at node 1, with the user specifying the fragment number.

Method:
For updating the value of fragment i at each node, the user inputs the fragment, and the new value he needs to update it to.

1. We initialize message of first node with tuple containing index of fragment that needs to be updated and its updated value. All other nodes are initialized with tuple containing 0.

2. Whenever node receives message it checks for first value of tuple of message and if it has non zero value then message is update message. We will then check whether we have corresponding index value in fragment. If we have that index then we will update fragment with new value. Otherwise we will wait for messages.

Results/Graphs:

![Graph](image-url)

NT=Neighborhood Threshold, RF= Replication Factor, Error=Acceptable Error(%), No. of Msgrs to reach convergence with defined params
Conclusion:
We have considered no error in the update function. Which means that the updated value should reach to all the nodes. As can be seen from the graph, this is experimentally proven as well.

5) Retrieve: Retrieve the contents of fragment i from any node that may have an up-to-date copy of it. The retrieval is requested by node 1, with the user specifying the fragment number.

Assumptions:
1. Retrieval originates from Node 1 with user specifying the index of the value to be retrieved.
2. The network is stable for the length of running gossip, at least one path exists from the node at which the index was found to Node 1 where the retrieval was requested.
3. As part of second assumption, Node 1 stays alive for entire length of gossip.

Method:
User specifies the index of value to be updated as input. After spawning all the threads, the process id(pid) of Node 1 is found out and is included as part of the message that will be sent out from Node 1. Other nodes will have a default message [0, 0] which denotes that they haven’t got the index value to be updated yet.

After gossip process has been started, 3 types of interactions take place based on the message nodes are exchanging and the fragments they have as explained in the following step. X and Y are two nodes talking to each other. Update message means the message containing index value to be retrieved and Pid of first node.

1. If X is sending update message and Y didn’t have update message before talking with X, it stores Pid of X in a variable.
2. If Y had the update message already or if the message sent by X didn’t have the update message it doesn’t make any change.
3. In case X didn’t have update message initially and it got the update message from Y, it does the same steps as explained first.

When a node gets the update message for the first time, it checks whether the fragment it owns has the corresponding index value. If not it updates its message and passes it on. If it finds the index value in the fragment, it stores the value in the message and sends a message to the node it got message from. The sender node upon receiving message from receiver node, passes the message to it’s the node which passed it’s update message(whose Pid was stored in variable as explained in step 1). Thus it traces back the path from where the value was found and when Pid of a node equals to the Pid stored in the message(i.e of first node) the retrieval process stops and it prints out the value.

Results/Conclusions:
For a given number of nodes with a randomly generated topology, retrieve was successful and very efficient. We realize that in real world networks the stability of the path built while gossiping the update message cannot guaranteed throughout the process. Chances are that nodes belonging to the path might fail and therefore retrieval value cannot be returned to parent node. For overcoming this problem we would have to restart the retrieval process once Gossip has been found to be unresponsive. This would also require to determine the time interval after which we should judge the computation to be failed.

Alternative Method for Retrieve: The other solution we had at hand was to gossip the retrieved value back to the first node but this would make the retrieval process very slow and inefficient and therefore was overlooked.
Future Work:

1) We can implement quorums for retrieve of value of index. This will make algorithm robust and it would be enable us to handle failure.
2) If given more time, I would like to add feature to framework for adding and deleting (failing) of node. If node joins it gets attached to node which is already in network, its neighbors list will increase and will have both the nodes.

References:

[8] www.erlang.org
MAXIMUM

Fragments = \{1, 2, 3, 4, 5, 6\}

Initialization of Nodes:

List of fragments with Node 1 = \{2, 3, 4\}. MyValue = 4.

Notation Used:-

\[
\begin{array}{c}
\text{Node} \\
\text{MyValue} \\
4
\end{array}
\]

Gossip Phase:-

\[
\begin{array}{c}
1 \\
\text{MyValue} = 4 \\
4
\end{array} \quad \begin{array}{c}
2 \\
\text{MyValue} = 3 \\
3
\end{array} \quad \begin{array}{c}
1 \\
4
\end{array} \quad \begin{array}{c}
2 \\
4
\end{array}
\]

Initial State                Final State

\[
\begin{array}{c}
1 \\
\text{MyValue} = 4 \\
4
\end{array} \quad \begin{array}{c}
4 \\
\text{MyValue} = 6 \\
6
\end{array} \quad \begin{array}{c}
1 \\
6
\end{array} \quad \begin{array}{c}
4 \\
6
\end{array}
\]

Initial State                Final State
MINIMUM

Fragments = \{1, 2, 3, 4, 5, 6\}

Initialization of Nodes:

List of fragments with Node 1 = \{2, 3, 4\}. MyValue = 2.

Notation Used:

![Diagram of Node and MyValue]

Gossip Phase:

![Diagram of gossip phase]

Initial State          Final State

Initial State          Final State
AVERAGE

Node 1: Fragments = \{2, 4, 6\}

Node 2: Fragments = \{5, 7, 3, 4, 1\}

Notation Used:-

<table>
<thead>
<tr>
<th>Node</th>
<th>Length</th>
<th>Sum</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>16</td>
<td>4</td>
</tr>
</tbody>
</table>

We can see that in final state, Average = Sum/Length = 4 = actual average of all the fragments.
<table>
<thead>
<tr>
<th>Process</th>
<th>Count</th>
<th>Bytes</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>lo:wait_lo_mon_reply/2</td>
<td>20</td>
<td>6765</td>
<td>11175 0</td>
</tr>
<tr>
<td>net_kernel:spawn_func/6</td>
<td>4181</td>
<td>14362</td>
<td>1</td>
</tr>
<tr>
<td>lo:wait_lo_mon_reply/2</td>
<td>20</td>
<td>4181</td>
<td>18158 1</td>
</tr>
<tr>
<td>net_kernel:spawn_func/6</td>
<td>4181</td>
<td>13777</td>
<td>2</td>
</tr>
<tr>
<td>lo:wait_lo_mon_reply/2</td>
<td>20</td>
<td>4181</td>
<td>13879 1</td>
</tr>
<tr>
<td>net_kernel:spawn_func/6</td>
<td>4181</td>
<td>12560</td>
<td>1</td>
</tr>
<tr>
<td>lo:wait_lo_mon_reply/2</td>
<td>20</td>
<td>4181</td>
<td>11147 0</td>
</tr>
<tr>
<td>net_kernel:spawn_func/6</td>
<td>4181</td>
<td>14255</td>
<td>1</td>
</tr>
<tr>
<td>lo:wait_lo_mon_reply/2</td>
<td>20</td>
<td>4181</td>
<td>16534 1</td>
</tr>
<tr>
<td>net_kernel:spawn_func/6</td>
<td>4181</td>
<td>13383</td>
<td>0</td>
</tr>
<tr>
<td>lo:wait_lo_mon_reply/2</td>
<td>20</td>
<td>4181</td>
<td>12977 1</td>
</tr>
<tr>
<td>net_kernel:spawn_func/6</td>
<td>4181</td>
<td>13252</td>
<td>2</td>
</tr>
<tr>
<td>lo:wait_lo_mon_reply/2</td>
<td>20</td>
<td>4181</td>
<td>11336 0</td>
</tr>
<tr>
<td>net_kernel:spawn_func/6</td>
<td>4181</td>
<td>13984</td>
<td>0</td>
</tr>
<tr>
<td>lo:wait_lo_mon_reply/2</td>
<td>20</td>
<td>4181</td>
<td>15776 0</td>
</tr>
<tr>
<td>net_kernel:spawn_func/6</td>
<td>4181</td>
<td>11463</td>
<td>1</td>
</tr>
<tr>
<td>lo:wait_lo_mon_reply/2</td>
<td>20</td>
<td>4181</td>
<td>13265 1</td>
</tr>
<tr>
<td>net_kernel:spawn_func/6</td>
<td>4181</td>
<td>13536</td>
<td>0</td>
</tr>
</tbody>
</table>
APPENDIX C.

This appendix contains the results that were used to display the values used for the various graphs throughout the paper.

Notations Used:

RF = Replication Factor.  
Nodes = No. of Nodes. 
NT = Neighbourhood Threshold.  
ACC = Acceptable Error.

### Minimum RF=1, Acc=0.01

<table>
<thead>
<tr>
<th>Neighborhood Threshold</th>
<th>100 Nodes</th>
<th>500 Nodes</th>
<th>1000 Nodes</th>
<th>1500 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>NT=0.2</td>
<td>4755</td>
<td>21013</td>
<td>43430</td>
<td>92728</td>
</tr>
<tr>
<td>NT=0.3</td>
<td>3055</td>
<td>18786</td>
<td>40364</td>
<td>85659</td>
</tr>
</tbody>
</table>

### Maximum RF=1, Acc=0.01

<table>
<thead>
<tr>
<th>Neighborhood Threshold</th>
<th>100 Nodes</th>
<th>500 Nodes</th>
<th>1000 Nodes</th>
<th>1500 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>NT=0.2</td>
<td>4143</td>
<td>16046</td>
<td>42237</td>
<td>66588</td>
</tr>
<tr>
<td>NT=0.3</td>
<td>3618.5</td>
<td>15370</td>
<td>38041</td>
<td>65194</td>
</tr>
</tbody>
</table>

### Maximum NT=0.2 Acc=0.1

<table>
<thead>
<tr>
<th>Replication Factor</th>
<th>100 Nodes</th>
<th>500 Nodes</th>
<th>1000 Nodes</th>
<th>1500 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>RF=1</td>
<td>3839</td>
<td>17064</td>
<td>41006</td>
<td>63257</td>
</tr>
<tr>
<td>RF=2</td>
<td>2390</td>
<td>12733</td>
<td>34040</td>
<td>62201</td>
</tr>
<tr>
<td>RF=3</td>
<td>2572</td>
<td>13971</td>
<td>29999</td>
<td>52236</td>
</tr>
</tbody>
</table>

### Minimum NT=0.2 Acc=0.1

<table>
<thead>
<tr>
<th>Replication Factor</th>
<th>100 Nodes</th>
<th>500 Nodes</th>
<th>1000 Nodes</th>
<th>1500 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>RF=1</td>
<td>5170</td>
<td>19268</td>
<td>49554</td>
<td>71933</td>
</tr>
<tr>
<td>RF=2</td>
<td>3866</td>
<td>21573</td>
<td>38717</td>
<td>65342</td>
</tr>
<tr>
<td>RF=3</td>
<td>5359</td>
<td>19839</td>
<td>35949</td>
<td>62715</td>
</tr>
</tbody>
</table>
### Average NT=0.2 Acc=0.1

<table>
<thead>
<tr>
<th>Replication Factor</th>
<th>100 Nodes</th>
<th>500 Nodes</th>
<th>1000 Nodes</th>
<th>1500 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>RF=1</td>
<td>128477</td>
<td>245536</td>
<td>486086</td>
<td>1826759</td>
</tr>
<tr>
<td>RF=2</td>
<td>46045</td>
<td>176449</td>
<td>274075</td>
<td>565421</td>
</tr>
<tr>
<td>RF=3</td>
<td>46581</td>
<td>155599</td>
<td>260489</td>
<td>555527</td>
</tr>
</tbody>
</table>

### Average NT=0.2 RF=1

<table>
<thead>
<tr>
<th>Acceptable Error(%)</th>
<th>100 Nodes</th>
<th>500 Nodes</th>
<th>1000 Nodes</th>
<th>1500 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC=0.1</td>
<td>128477</td>
<td>245536</td>
<td>486086</td>
<td>1826759</td>
</tr>
<tr>
<td>ACC=0.5</td>
<td>46045</td>
<td>176449</td>
<td>274075</td>
<td>565421</td>
</tr>
<tr>
<td>ACC=1</td>
<td>46581</td>
<td>155599</td>
<td>260489</td>
<td>555527</td>
</tr>
</tbody>
</table>

### Maximum NT=0.2 RF=1

<table>
<thead>
<tr>
<th>Acceptable Error(%)</th>
<th>100 Nodes</th>
<th>500 Nodes</th>
<th>1000 Nodes</th>
<th>1500 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4264</td>
<td>18492</td>
<td>41022</td>
<td>71828</td>
</tr>
<tr>
<td>0.01</td>
<td>3642</td>
<td>17838</td>
<td>36484</td>
<td>55295</td>
</tr>
<tr>
<td>0.1</td>
<td>3642</td>
<td>13486</td>
<td>26510</td>
<td>51300</td>
</tr>
</tbody>
</table>

### Minimum NT=0.2 RF=1

<table>
<thead>
<tr>
<th>Acceptable Error(%)</th>
<th>100 Nodes</th>
<th>500 Nodes</th>
<th>1000 Nodes</th>
<th>1500 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4761</td>
<td>19991</td>
<td>46838</td>
<td>71933</td>
</tr>
<tr>
<td>0.01</td>
<td>4761</td>
<td>19991</td>
<td>46838</td>
<td>71933</td>
</tr>
<tr>
<td>0.1</td>
<td>4761</td>
<td>19991</td>
<td>46838</td>
<td>71933</td>
</tr>
</tbody>
</table>