UMBC CSEE DEPARTMENT

Project2

Advanced Operating System

Sandipan Dey
Ronil Mokashi
Kavita Dabke

12/14/2009

Problem Statement, Model and Assumptions

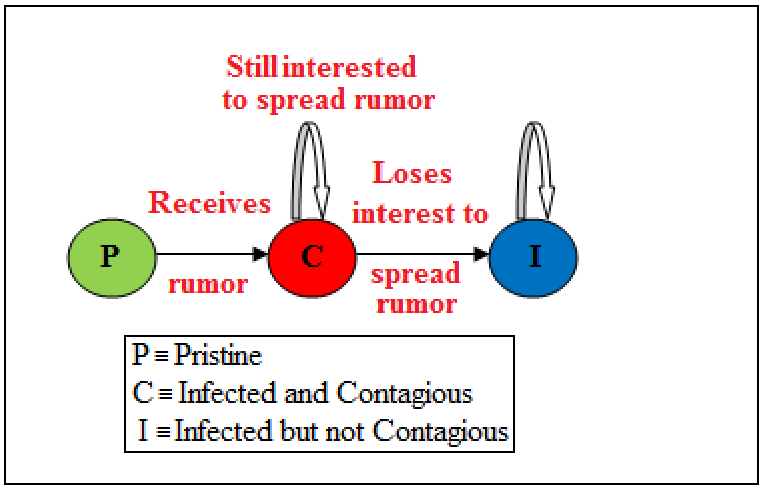
We have implemented the following algorithms in Erlang:

- Min / Max
- Average
- Distributed Outlier Detection
- Median

- ➢ We implemented the algorithms by **rumor-spreading** / epidemic / anti-entropy mechanism.

- ➢ We simulated the nodes by Erlang processes, i.e. each node represent one **Erlang process**.

- ➢ At any time, each node can typically be in any of the following 3 states:
  1) **Pristine** (NOT infected)
  2) **Contagious** (Infected and willing to infect others by spreading rumor).
  3) **Infected** (already infected but no longer interested to spread rumor).


Still interested to spread rumor
Receives rumor
Loses interest to spread rumor
P
C
I
P ≡ Pristine
C ≡ Infected and Contagious
I ≡ Infected but not Contagious

State transition diagram of a node (Erlang process)

- ➢ Each node will have some initial **value** to start with. Our objective is to compute distributed aggregates via gossiping out of those initial data values.

- ➢ Initially all the nodes are **pristine**, except **one** that will be **contagious**.

- ➢ We run the iterative algorithm for computing the distributed aggregate by spreading the rumor. The algorithm is described in figure-1.

- ➢ In each iteration, a contagious node (if any) will spread the infection / rumor to a randomly selected node. For random selection we used **uniform pseudo-random generator** provided by Erlang, with proper **initialization** of the **seed** (e.g., by **current time**). This step is very important and has a large impact in convergence of the gossip algorithm, since we need to ensure that the same subset of nodes are not selected every time and almost all (if not all) nodes are reached (little or very few nodes are left out). Another alternative technique we used instead of this pseudo random generator is **random permutation** technique, obtained by using

randomized partition technique**,** to guarantee that a contagious node is **not** choosing the **same neighbor** at any two iterations to gossip**.**

- Any node that receives an (update) message (with its value) from a contagious node, will immediately re-compute its own value by taking aggregate measure with the one that it received. It will send the aggregate it computed back to the node it received the message from.

- For **Min**/**Max**/**Average** computation, the **rumor** that a contagious node will wants to **spread** will be its own **value** present at the node, for **outlier detection** the rumor a contagious node wants to spread will be the **local top-k outliers** computed by the node itself.

- This process is guaranteed to converge since at each iteration some aggregate is computed that is guaranteed to be closer (or at same distance, either will improve or no change) to the actual aggregate. For instance when computing the **average**, we shall have the following always true: $\min(v_i, v_j) \leq v_{ij} \leq \max(v_i, v_j)$.

- For aggregates like min/max/ average, each node will hold a value that will be stored in process's memory, but for outlier detection, since each node may have a huge **multi-dimensional** dataset, we store the data locally to each node (in a separate directory for each Erlang process) in disk files, which we read from / write into while gossiping.

- We used Erlang gs package to graphically plot the data values for each node.

How to run the programs

gossip.erl / gossip1.erl / gossip2.erl

gossip: runAlgo (number_of_nodes, algo_name, convergence_factor).

e.g.,

```
erl
c (gossip).
gossip: runAlgo (1000, avg, 4).
```

will run the program for N = 1000 nodes, to compute distributed average, for 4 * log(N) iterations.

outlier.erl (top k outliers)

outlier: runAlgo(number_of_nodes, num_rows, num_cols, k, convergence factor).

e.g.,

```
outlier: runAlgo(100, 1000, 10, 5, 6).
```

will run the program for N = 100 nodes, to compute distributed top-5 global outliers, where each node will have 1000 x 10 data tuples, for 6 * log(N) iterations.

Algorithm Calculate distributed agreegates (min/max/average) via gossip/rumor spreading

Inputs
(1) $\aleph = \{n_1, n_2, \ldots, n_N\}$: set of N nodes, each node n_i with its initial value $v_i \in \Re$.
(2) $\lambda_{threshold}$: threshold to measure accuracy of computed aggregate.
(3) $stop_{threshold}$: probability with which a contagious node stops spreading infection.

Output: aggregate to be computed (e.g., for average computation, $v = \frac{1}{N}\sum_{i=1}^{N} v_i$, for minimum, $v = \underset{i}{min}(v_i)$ etc. and after convergence $\forall i,\ v_i \approx v$).

```
1: ∀$n_i \in \aleph$, $state(n_i) \leftarrow pristine$.
2: $state(n_1) \leftarrow contagious$.
3: do steps $4 - 17$ parallely at each node until convergence (no change in value at
   each node).
4: if $state(n_i) = contagious$ then
5:     $j \leftarrow random(N)$.
6:     send $v_i$ to the node $j$ {spread infection / rumor}.
7:     receive the aggregate value $v_{ij}$ from node $j$.
8:     $v_i \leftarrow v_{ij}$.
9:     if $|v_{ij} - v_i| < \lambda_{threshold}$ then {check if it already knows the rumor}
10:        $state(n_i) \leftarrow infected$ with probability $stop_{threshold}$. {no longer interested
           to spread the rumor}.
11:    end if
12: end if
13: if node $j$ receives $v_i$ from node $i$ then
14:    compute aggregate $v_{ij}$ from $v_i$ and $v_j$. {e.g., for average computation, $v_{ij} \leftarrow$
       $\frac{v_i+v_j}{2}$, for maximum value computation $v_{ij} \leftarrow \max(v_i, v_j)$.
15:    $v_j \leftarrow v_{ij}$.
16:    send $v_{ij}$ as reply back to node $i$.
17: end if
```

Figure -1: the algorithm

<u>Note on the convergence</u>

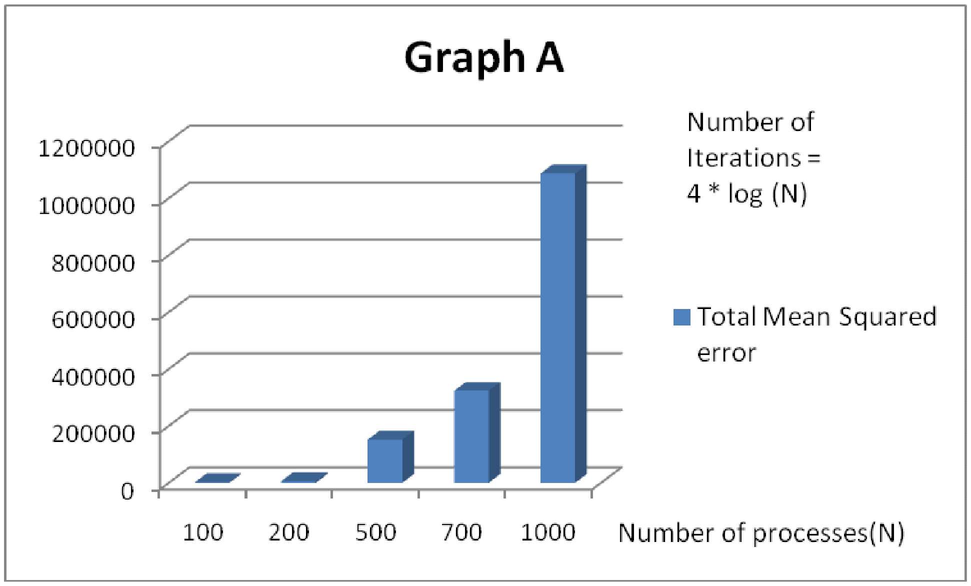
Since the gossip via (**epidemic**) rumor spreading is guaranteed to converge within $\theta(\lg n)$ steps and with $O(n \lg n)$ messages passed in between the nodes, we can use the number of iterations as the convergence criterion. In fact, we have tried with $C \times \lg(n)$ iterations with C a constant factor as the termination condition of our gossip algorithm.

Another useful measure for convergence criterion may be the **MSE** given by $\sum_{i=1}^{N}(v_{iF} - \bar{v})^2$, the mean square deviation of the actual aggregate from the computed ones at different nodes at convergence. We tried this metric too and have several results that are shown in the following graphs that show the impact of number of iterations on the convergence and on **MSE**.

Graph A, B, C shows the variation of the mean squared error (after convergence) with respect to number of nodes when the nodes contain data values from 1 to N, we repeat the experiments with different numbers of iterations. As can be seen, the variance of data also increases with the increase in number of nodes and as a result MSE increases drastically.

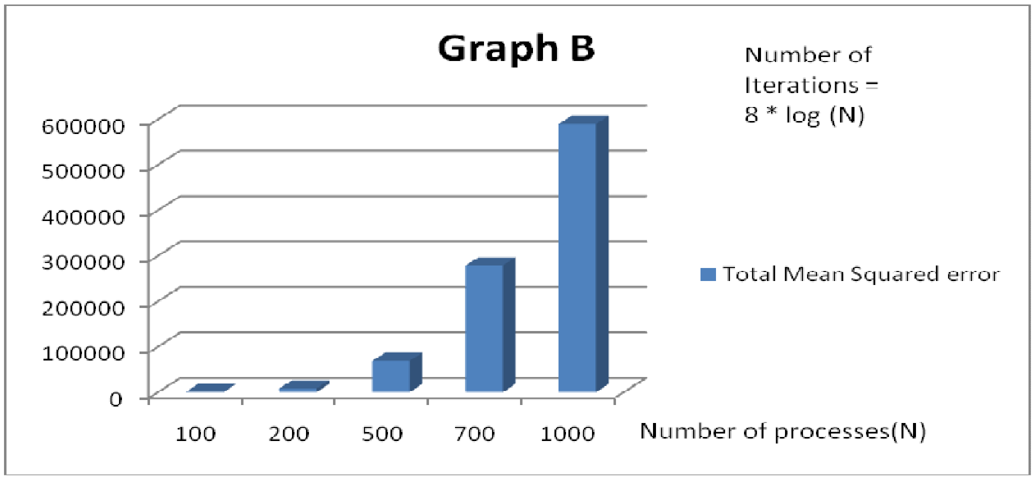
Effect of number of iterations on the mean squared error

Number of Iterations(4*log(N))	Number of processes	Total Mean Squared error
8	100	1696.971252
9	200	5874.234572
11	500	151684.3588
11	700	323034.1293
12	1000	1086756.611



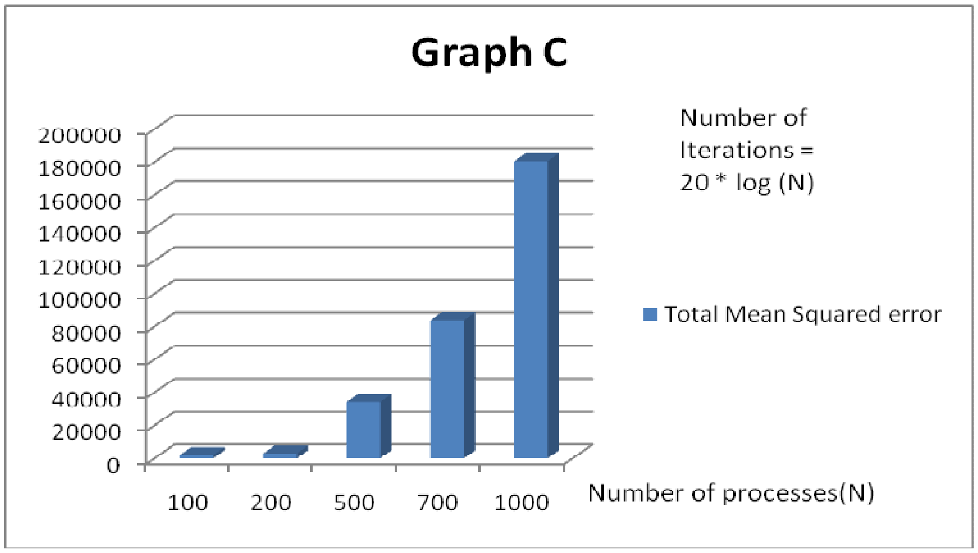
Effect of number of iterations on the mean squared error

Number of Iterations (8* log (N))	Number of processes	Total Mean Squared error
8	100	1017.435174
18	200	7998.184989
22	500	68912.77284
23	700	279108.2932
24	1000	589961.7374



Effect of number of iterations on the mean squared error

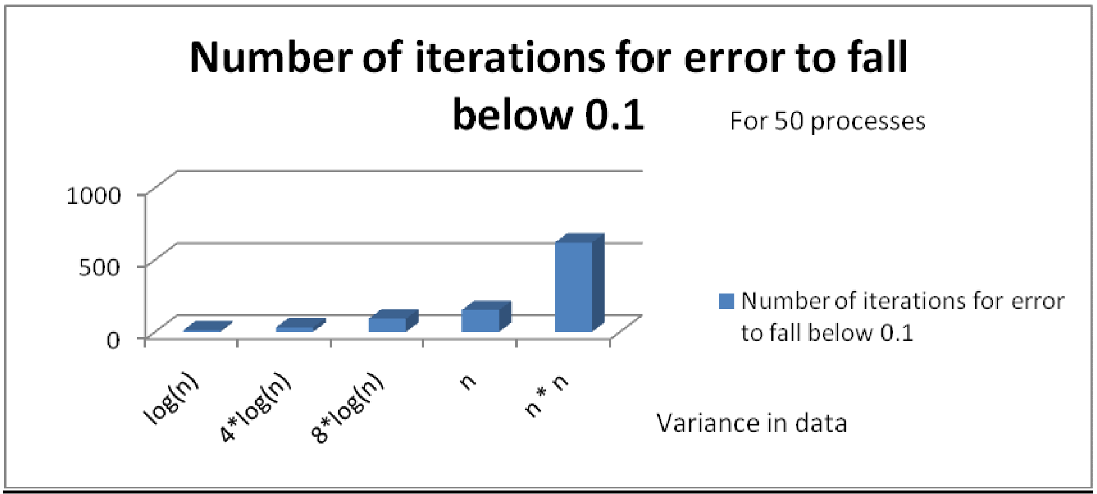
Number of Iterations(20*log(N))	Number of processes	Total Mean Squared error
40	100	1848.818187
46	200	2895.994794
54	500	34247.29527
57	700	83864.37009
60	1000	180438.4483



Effect of number of iterations on the mean squared error

Another important observation is that the impact of the variance in data values of the nodes on convergence / number of iterations. We did some experiments on it and here are the results.

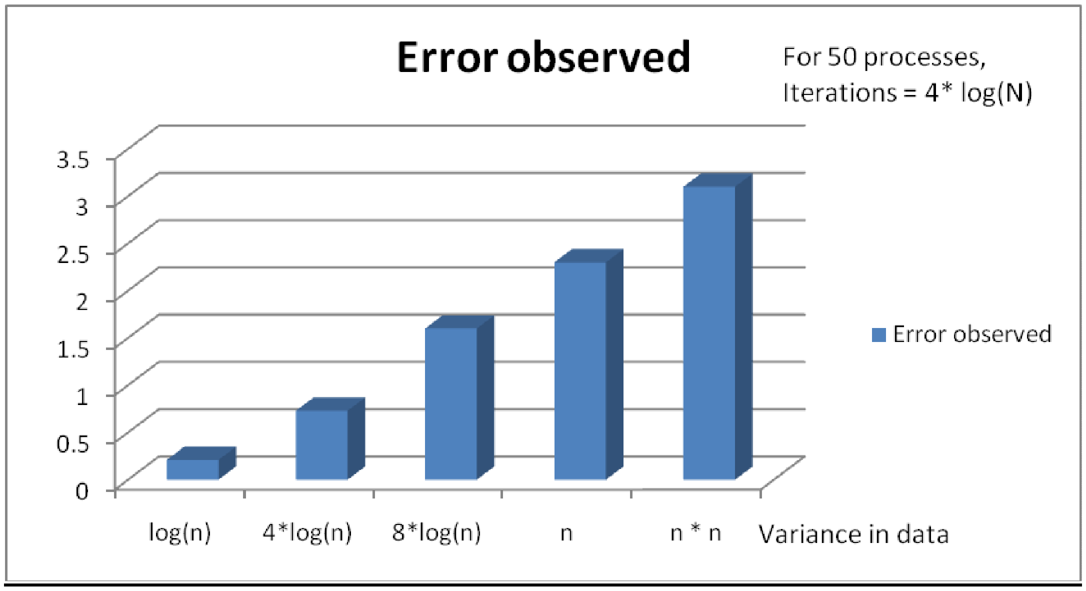
Number of processes	Range of data(variance)	Number of iterations for error to fall below 0.1
50	log(n)	10
50	4*log(n)	31
50	8*log(n)	92
50	n	152
50	n * n	620



The above result shows how many iterations are required to converge (make the MSE sufficiently small below a threshold, e.g., 0.1) with increasing variance in data. The data are genrated using uniform pseudo random genrator: initially the data generated with maximum range log(n), then with 4 * log(n) and in this way with a wide range of n * n with different n (e.g., with n = 50). Results show that when the variance of data increases the number of iterations to converge also increases drastically.

The next experiement is done keeping the number of iterations fixed, but increasing the variance of data and the MSE is observed after the gossip is over. As expected, there is a drastical increase in errors w.r.t. the increasing variance in data values.

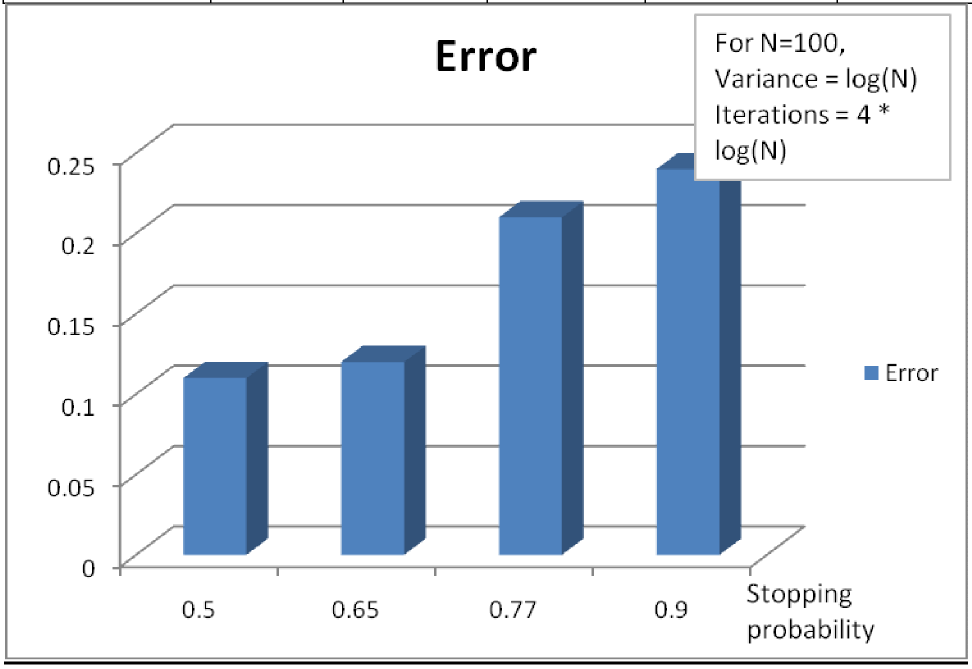
Number of processes	Number of iterations	Variance	Error observed
50	4*log(N)	log(n)	0.21
		4*log(n)	0.73
		8*log(n)	1.6
		n	2.3
		n * n	3.1

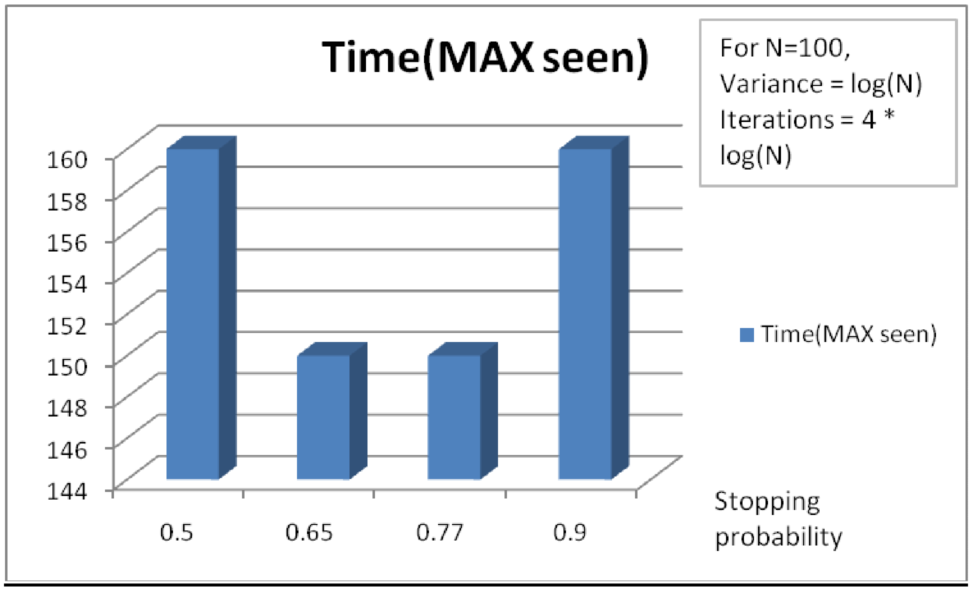


Here error is in terms of **MSE.**

Finally we did some experiments to show the variation of convergence time w.r.t. variation of the thresholds, results are shown below.

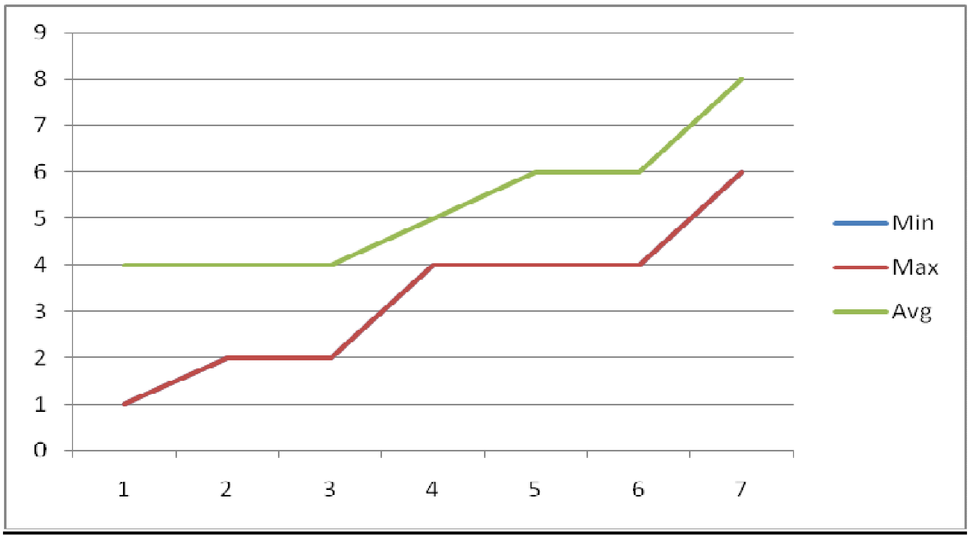
Number of processes	Variance	Iterations	Probability	Error	Time(MAX seen)
100	log(n)	4*log(n)	0.5	0.11	160
			0.65	0.12	150
			0.77	0.21	150
			0.9	0.24	160





We also observe that max/ min converges much faster than average.

	#iterations to converge (in log(n))		
#nodes	Min	Max	Avg
10	1	1	4
50	2	2	4
100	2	2	4
200	4	4	5
500	4	4	6
1000	4	4	6
10000	6	6	8



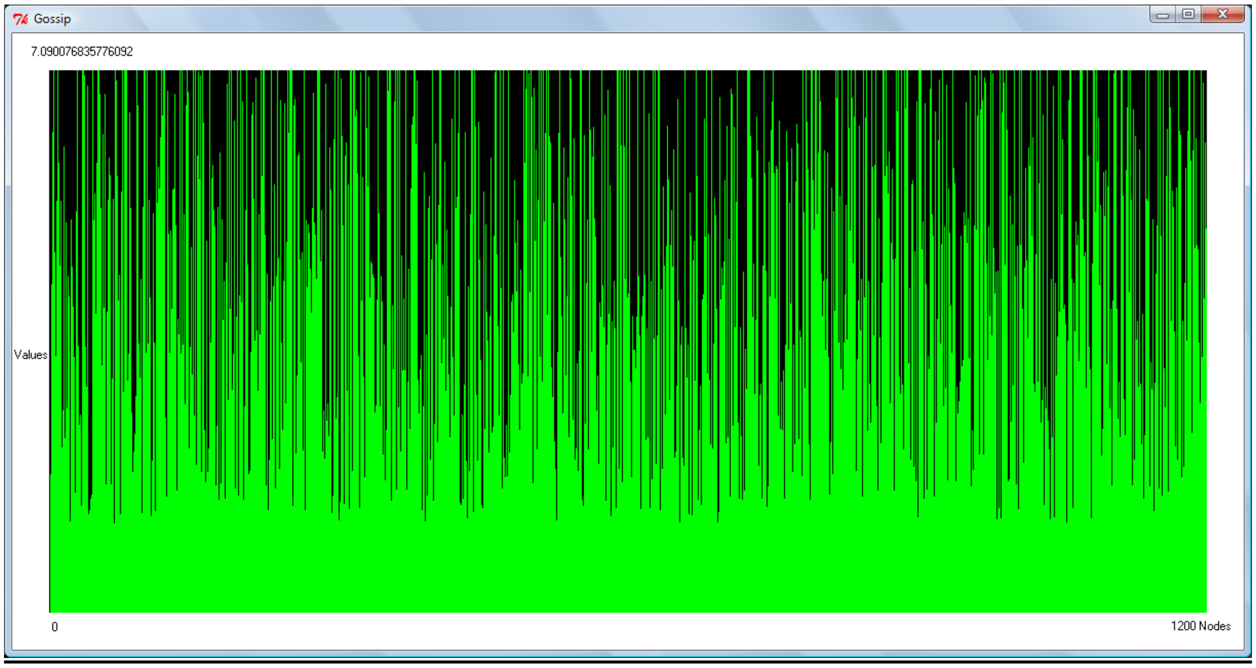
The Y axis represents #iterations * log(n)

Experimental Results / Screenshots

Computing Distributed Average / Min / Max aggregates

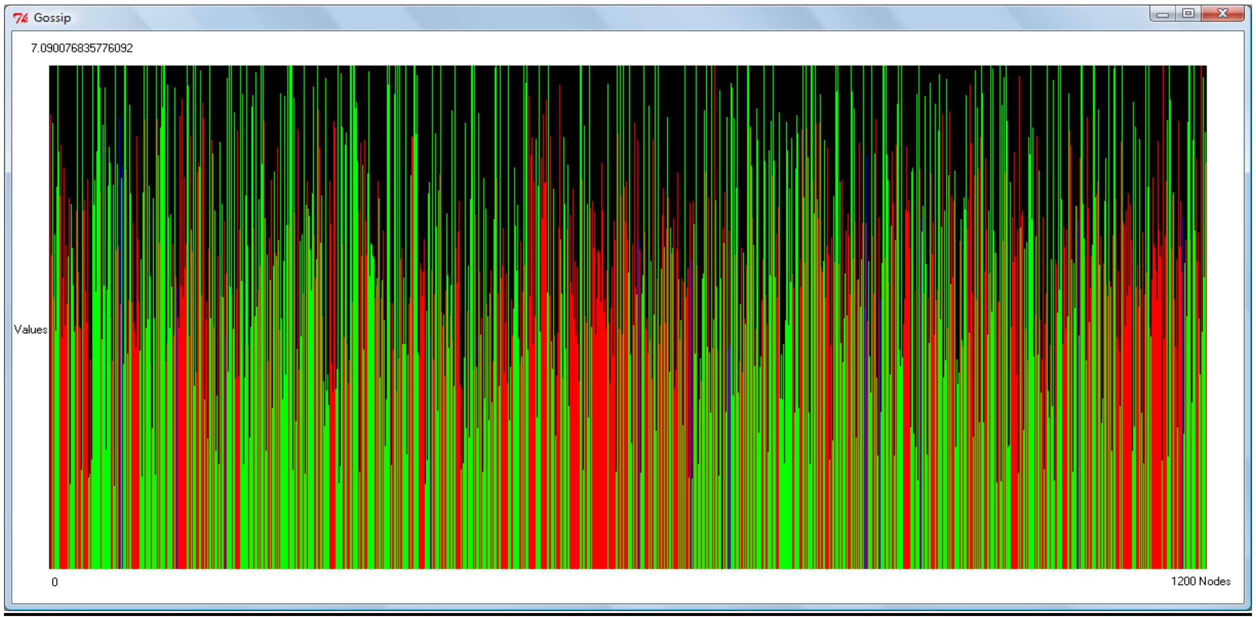
- Represents an experiment run with **1200** nodes.
- X-axis represents the nodes and Y axis represents the data values present at each of the node.
- Colors: Pristine Nodes: Green. Contagious Nodes: Red. Infected (but not contagious) nodes: Blue.

Initially all nodes are pristine. All the nodes are having data values in between 1 and 7.1 (values are randomly generated as well).



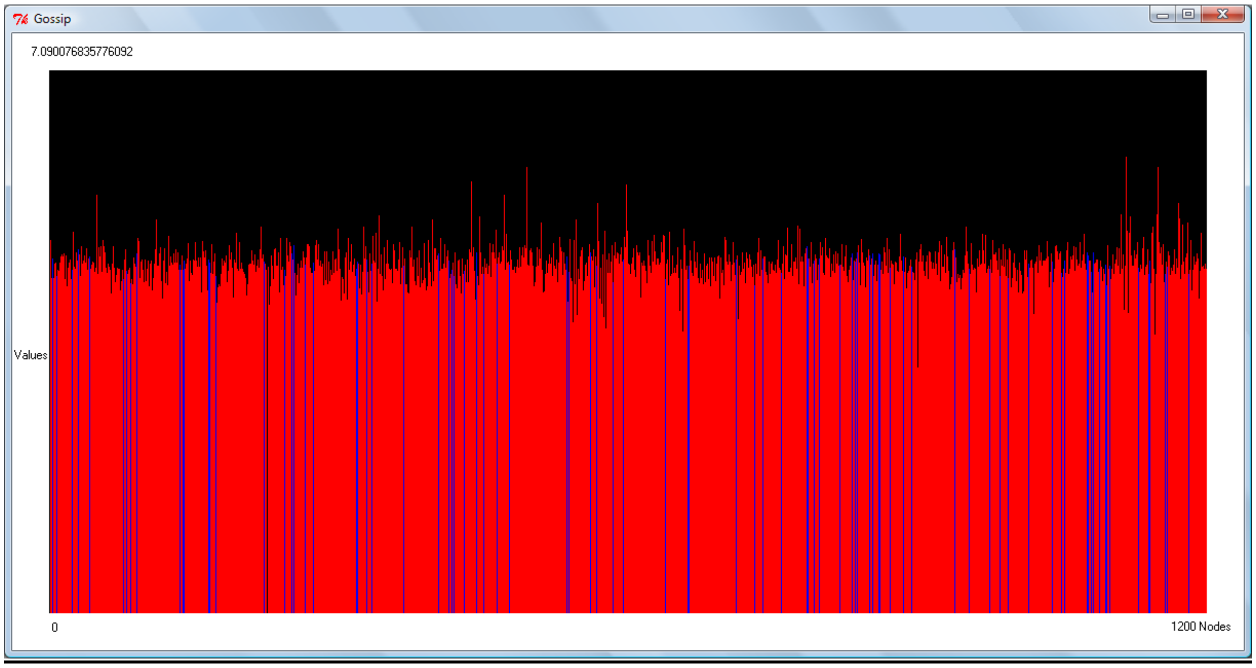
We then infect one node (i.e., change its state to **contagious**) and that particular node starts gossiping.

As the gossip algorithm continues, nodes become contagious.

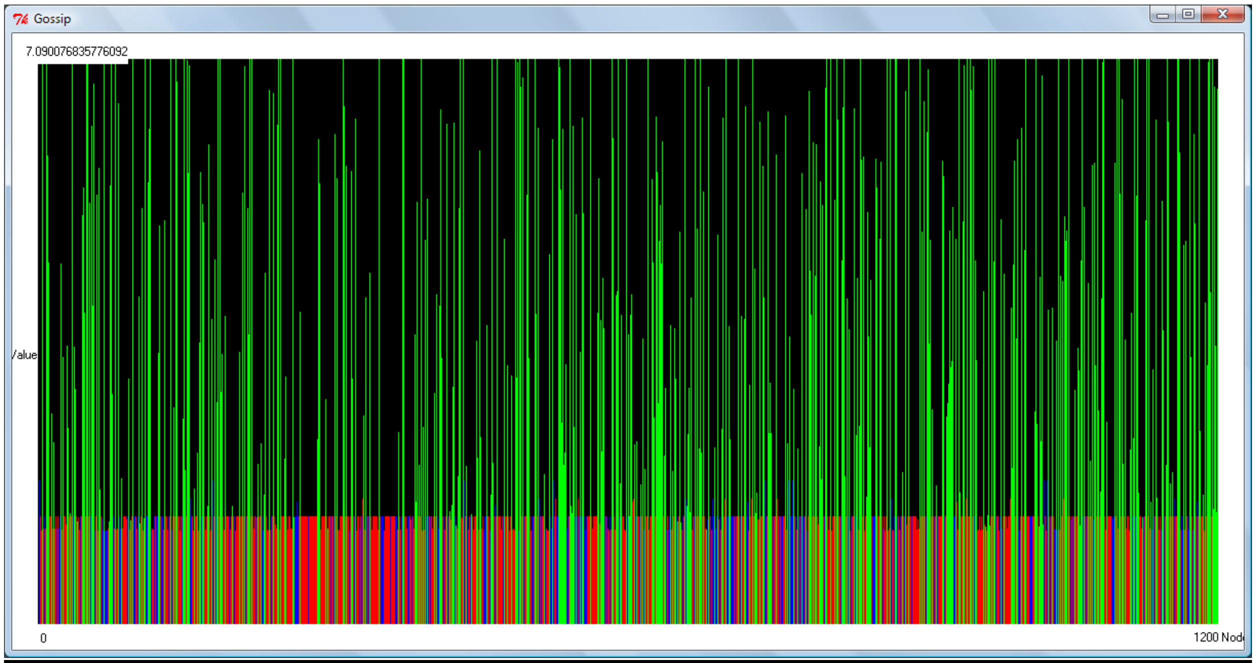


Gossiping average values

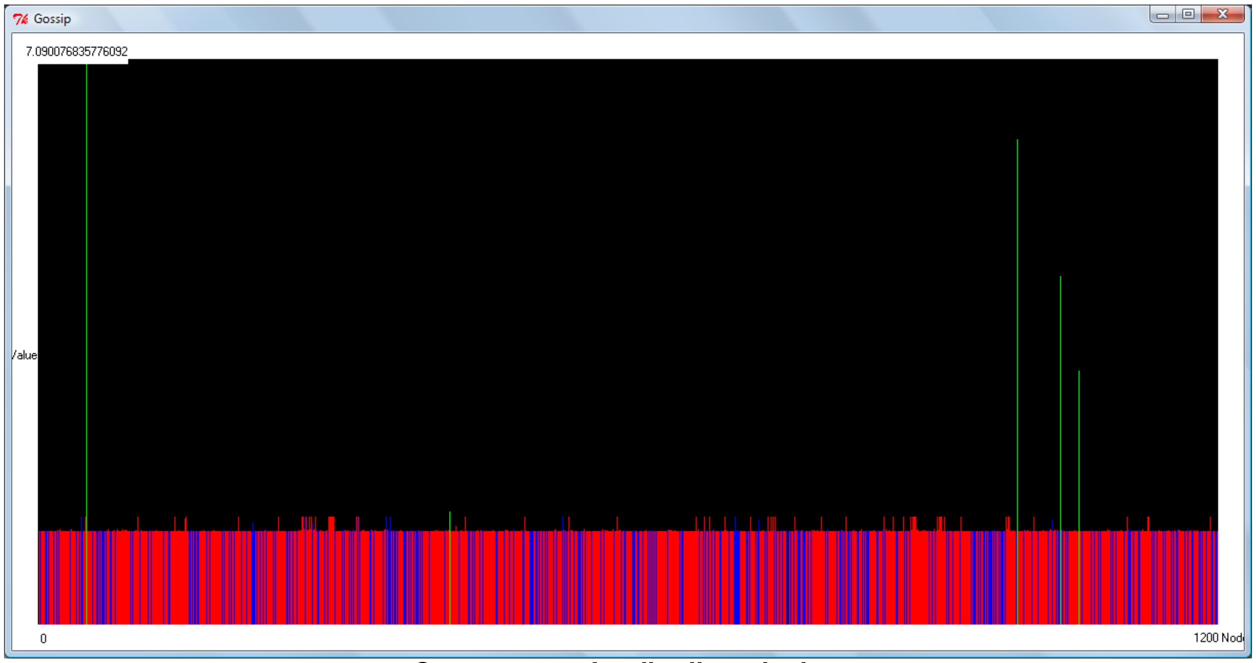
The following shows the data values of different nodes at convergence.
As can be seen, almost all the nodes are converged to the mean value. Another important thing to note is that at convergence all the nodes are either contagious or pristine and there is no node with pristine state (which implies that the final average computed has contribution from all the nodes).



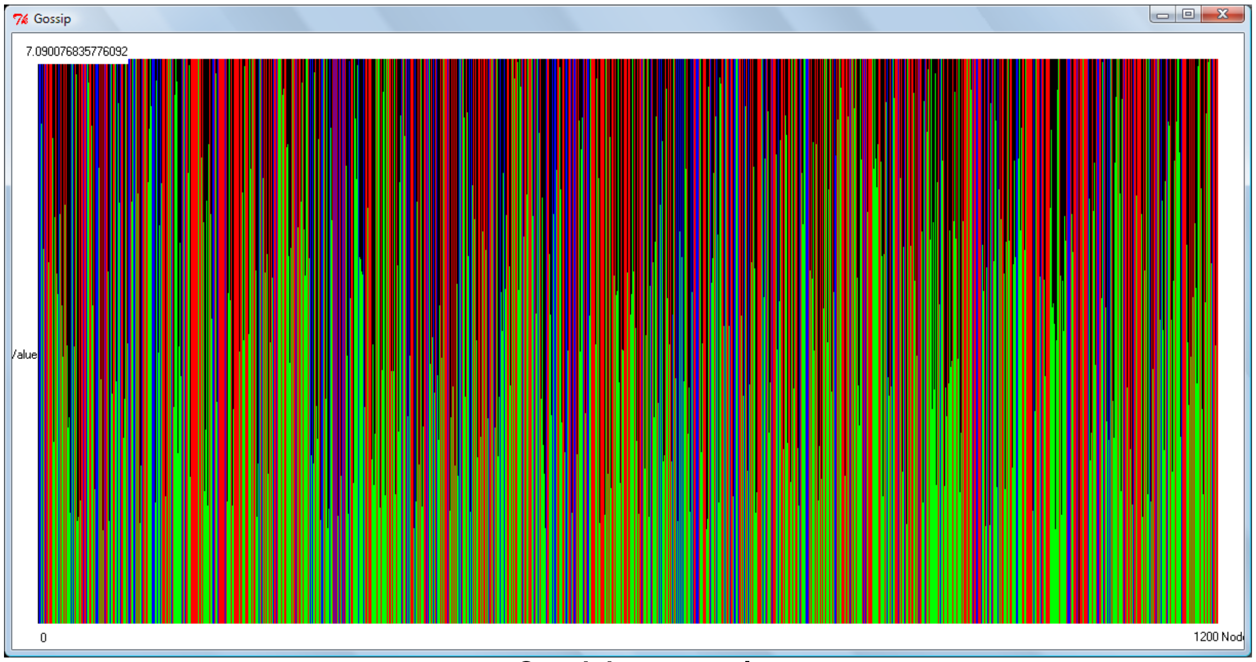
Convergence for distributed average



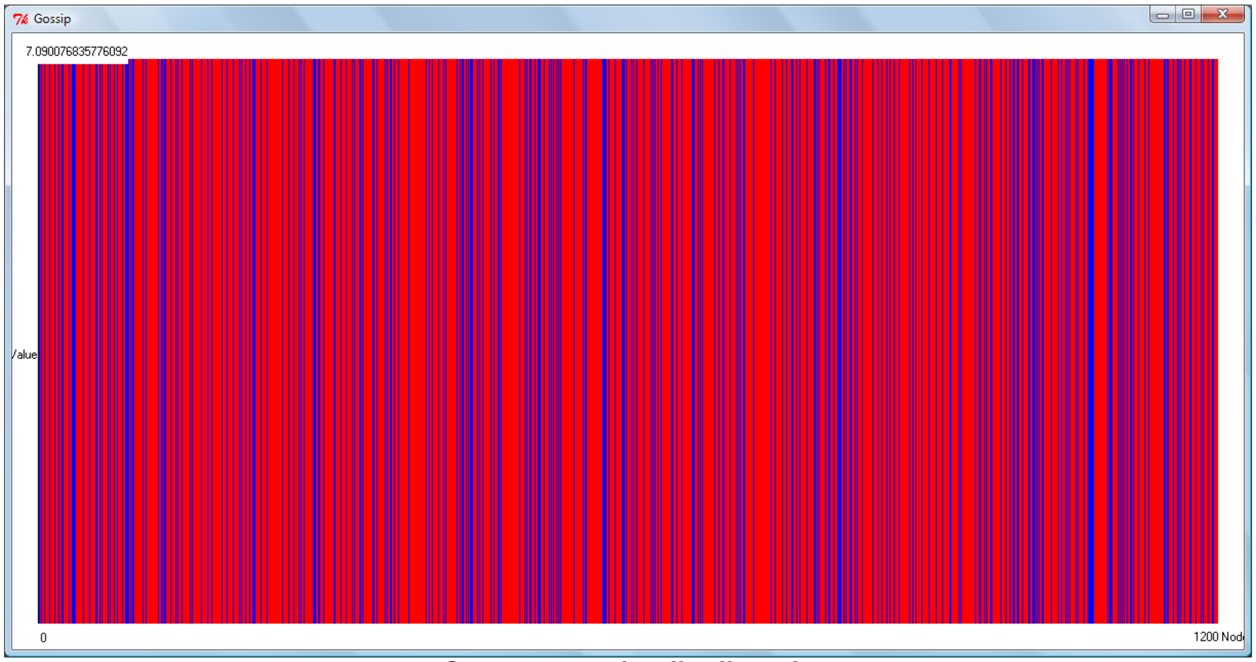
Gossiping min values



Convergence for distributed min



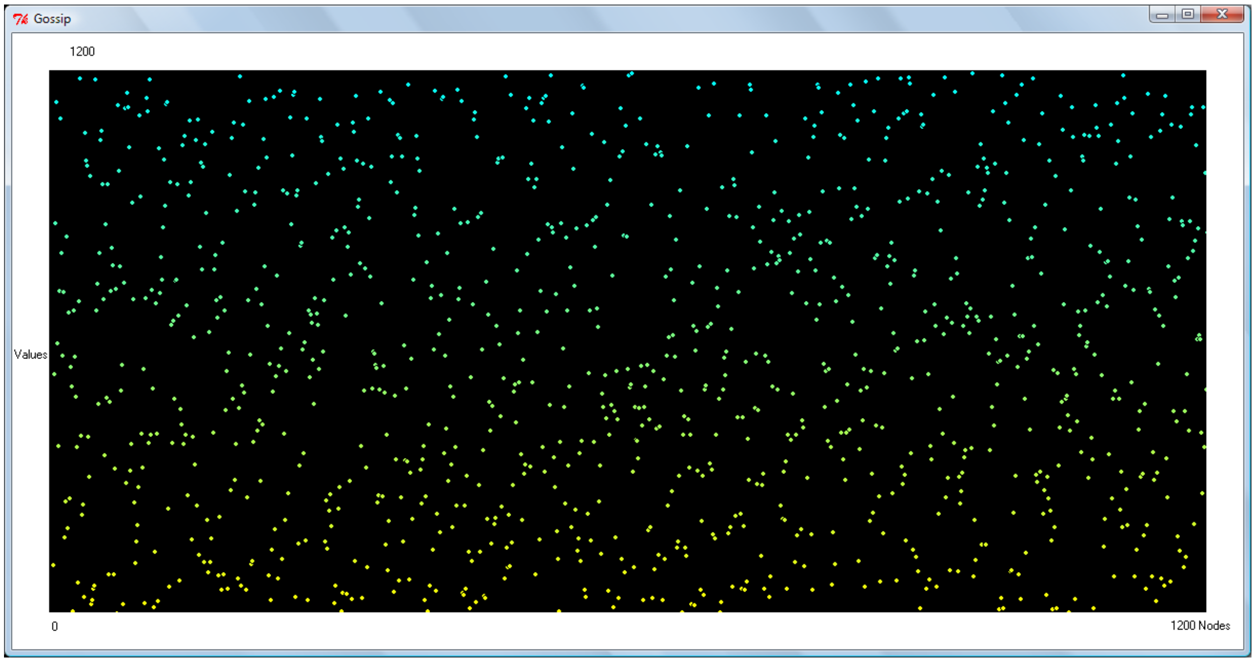
Gossiping max values

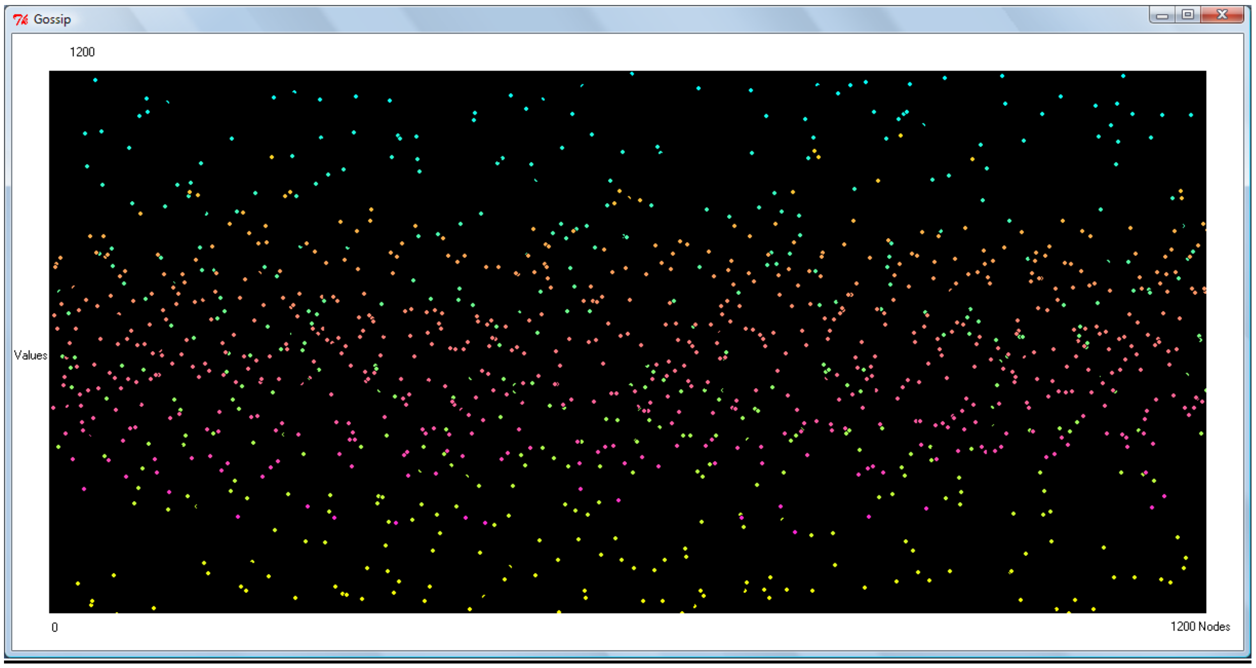


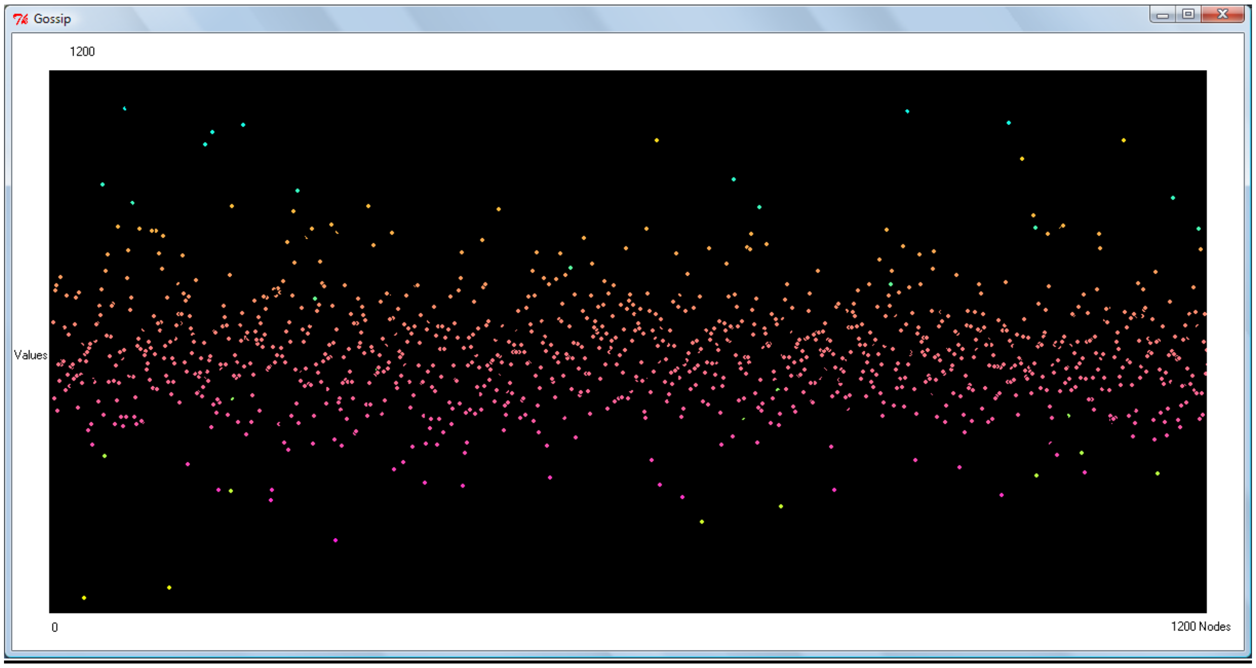
Convergence for distributed max

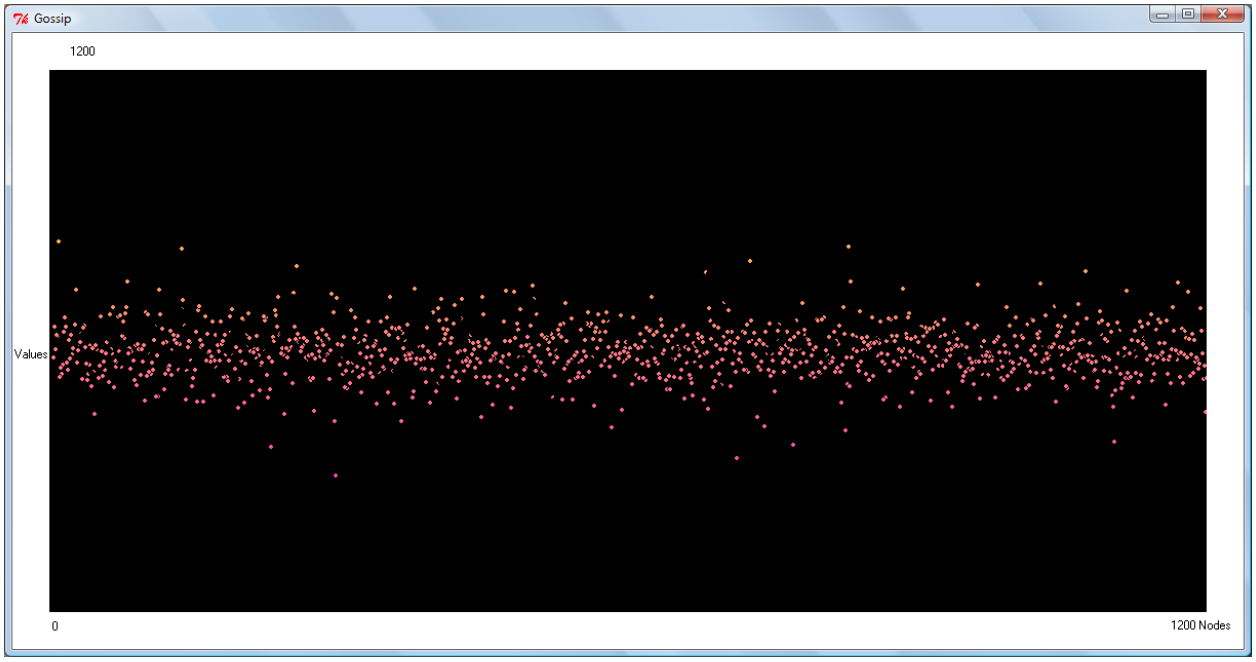
Another visualization of this convergence we had using point plots instead of line plot (with gradient color maps depending upon the data values at the points), but horizontal and vertical axes essentially representing the same entities as before (X Axis: Node number, Y Axis: Value at the node). Greenish nodes represent pristine nodes, while the reddish ones are contagious and infected nodes. The following figures show the convergence process.

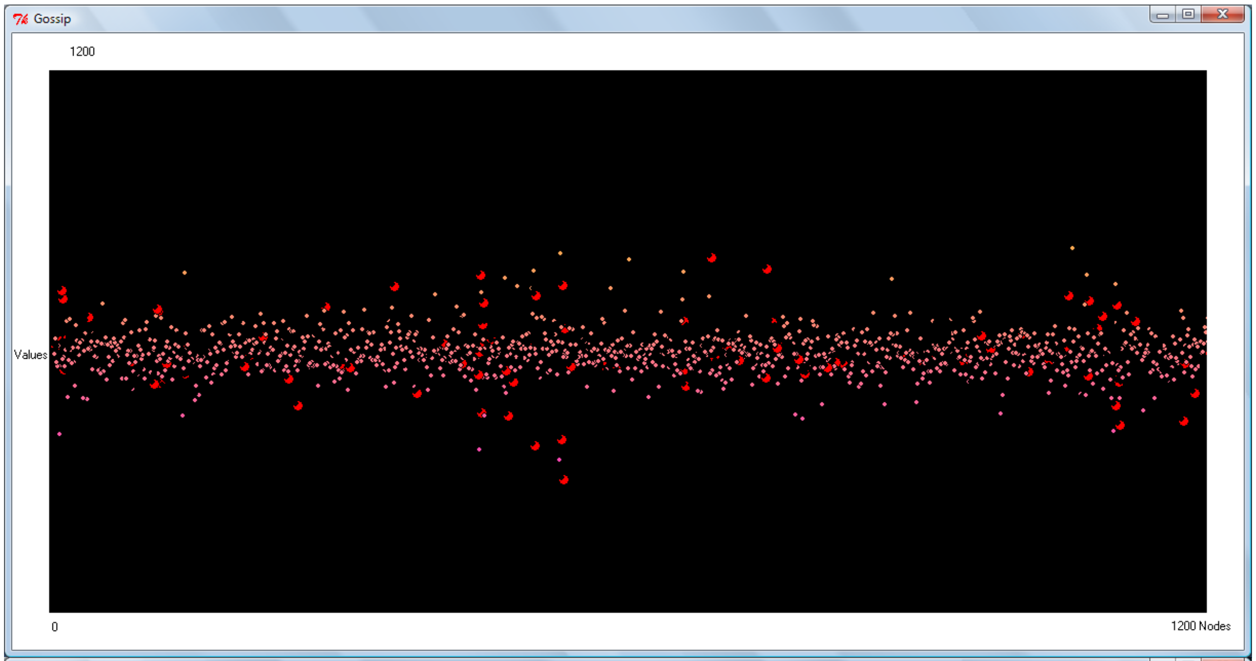
Here the data values at different nodes are from **1** to **1200** (data values generated randomly using uniform distribution).

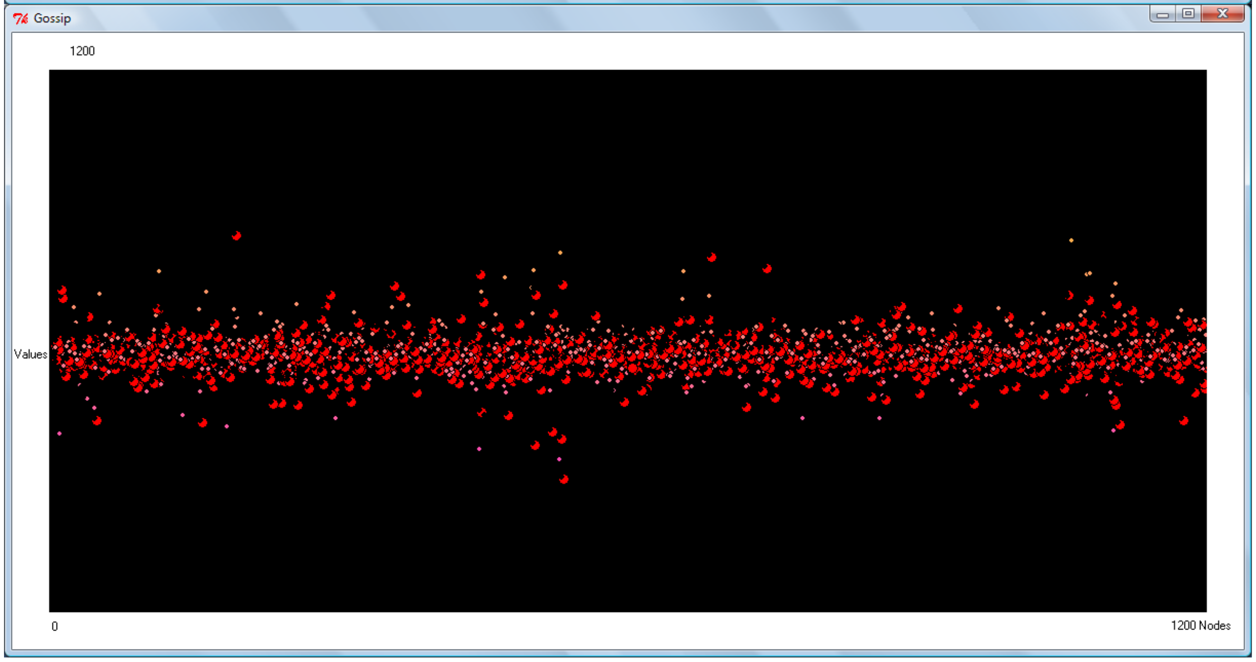

Gossip
1200
Values
0
1200 Nodes


Gossip
1200
Values
0
1200 Nodes


Gossip
1200
Values
0
1200 Nodes


Gossip
1200
Values
0
1200 Nodes

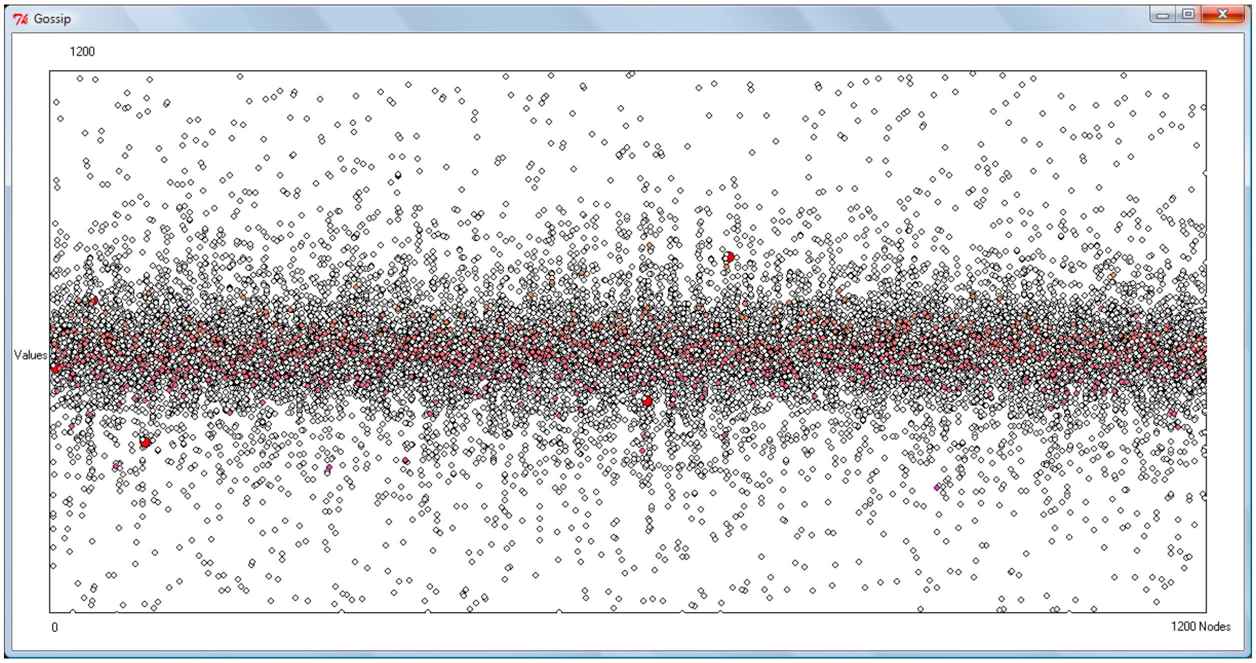




Here the big red circles represent the data values with which corresponding nodes converged (after $C \times \lg(n)$ iterations).

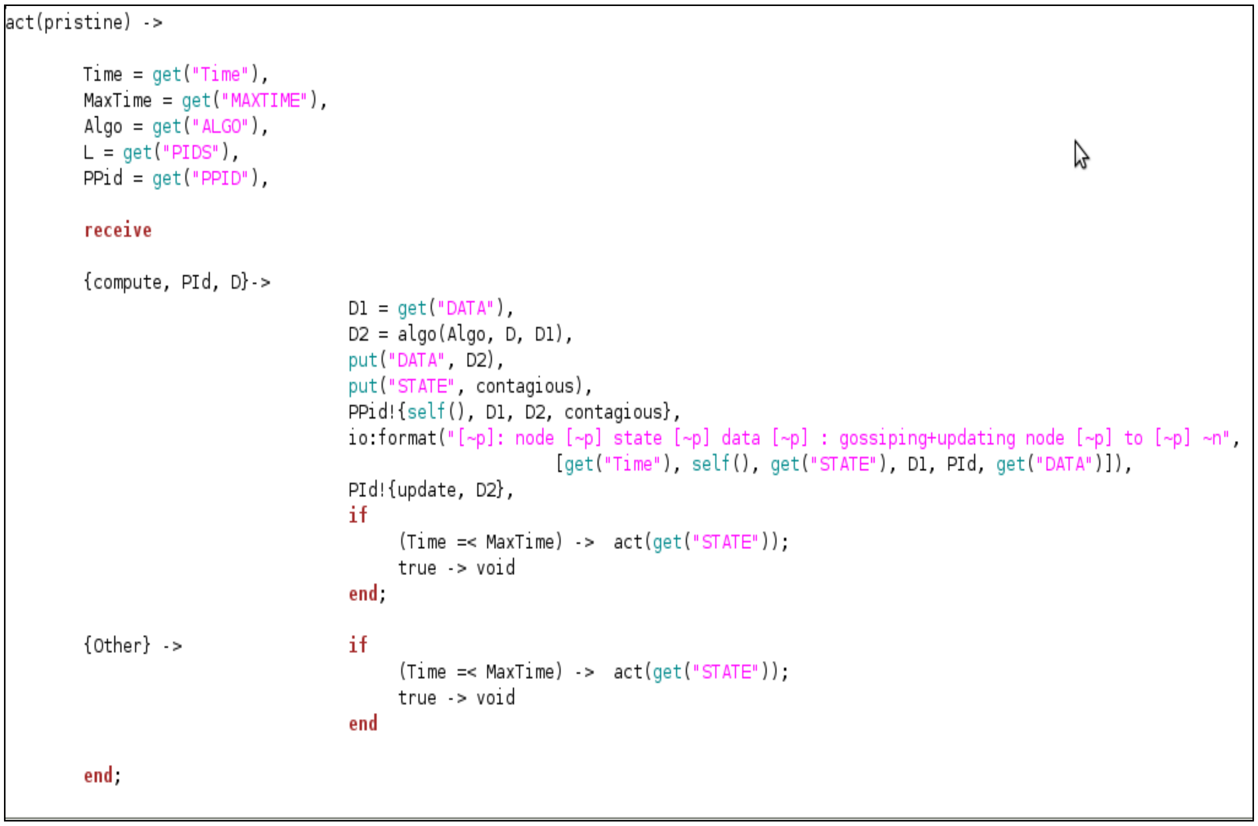
Another important observation is that the convergence rate decreases with time.

Yet another visual representation that shows the data values at the convergence for average computation:



The blank dots shows the earlier values, while the colored filled ones represent the data values at convergence while computing the distributed average.

The following figure shows a code fragment in Erlang:



```
act(pristine) ->

        Time = get("Time"),
        MaxTime = get("MAXTIME"),
        Algo = get("ALGO"),
        L = get("PIDS"),
        PPid = get("PPID"),

        receive

        {compute, PId, D}->
                                   D1 = get("DATA"),
                                   D2 = algo(Algo, D, D1),
                                   put("DATA", D2),
                                   put("STATE", contagious),
                                   PPid!{self(), D1, D2, contagious},
                                   io:format("[~p]: node [~p] state [~p] data [~p] : gossiping+updating node [~p] to [~p] ~n",
                                                        [get("Time"), self(), get("STATE"), D1, PId, get("DATA")]),
                                   PId!{update, D2},
                                   if
                                        (Time =< MaxTime) ->  act(get("STATE"));
                                        true -> void
                                   end;

        {Other} ->                 if
                                        (Time =< MaxTime) ->  act(get("STATE"));
                                        true -> void
                                   end

        end;
```

Distributed Outlier Detection (finding top-k global outliers) using gossip

Motivation

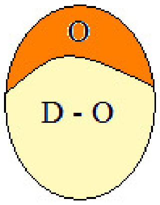
Outlier detection is one of the major tasks in data mining and has attracted attention since a long time. There can be many definations of the term outlier. In one way it can be defined as 'extreme data points' (noise?), but in some other way it can be defined as (usually a small) set of data points behaving anamolously w.r.t. the other data points. Offcourse, one needs to quantify the term 'anomolous behaviour'. That is usually defined by introducing notions of metrics: distance, density etc. and accordingly we have distance / density based methods for outlier detection. Also, we need to consider the neighborhood in which the point can be considered to be an outlier, interestingly the same point may not be considered as an abnormally behaving point if present in some other neighborhood. That brings the notion of local and global outliers. That brings the famous definition of distance-based outlier: if $p\%$ of the points are greater than d distance away from a given point, then that particular point can be treated as an outlier, with values of the given parameters p and d [5]. It means that the extreme (or noise) points are not the only candidates for being selected as outlier data points. Also, we must consider the fact that the outlier points may be scattered throughout the entire data space or they can together form a small set of outlier clusters as well. The outlier detection problem was also considered as neighborhood correlation problem [7]. Since our purpose is to divide the entire dataset into two parts, one being outliers, while the other being non-outlier points, the problem can be thought in terms of flow-CUT problem in graph theory as well [6] [3]. In this project we shall focus on posing the outlier detection problem as an optimization problem and fitting it into the distributed framework [2] [4] [1] to solve it in a distributed manner.

Before going to further details, let's first try to find the answer to a very basic question: why it is that important (if at all) to detect outliers in data? What if we leave them undetected? The answer to this question can probably be best answered when it is related to the domain of data. In general, there can be primarily two motivations for detecting outliers in data. First one is that the outliers themselves can represent interesting pattern in data that may lead us to answer some very important cause-effect relations in real world (e.g., sudden increase in traffic data at a point of time may signify an accident, while sudden increase in bank transaction data may be a candidate of a fraud) and in data mining our primary interest is to find hidden nontrivial useful patterns inside data (in a resource and computation efficient manner). Secondly, many of the existing data mining algorithms see outliers as noise points in data and the performance of them degrade due to presence of these outliers, thus require removal of outliers in preprocessing steps (e.g., in case of k-means clustering, if one of the outlier points is selected as centroid of a cluster). Hence, it's important to separate outlier points from the non-outlier points in the data set.

Problem Definition: Definition of Outliers

By definition, outliers are the data points that behaves anomolously w.r.t. other data points. Going in line with the usual deifinition, here we define the outliers as the data tuples that introduce more chaos in the data than other data tuples.
The chaos in the data is best measured by entropy of the data, which is defined as:
$H(X) = -\sum_{x \in \chi} p(x) log(p(x))$.
If we have the data set D and we are interested to find an outlier set O(containing top k outliers, where k is a user-chosen parameter) which will be asubset of the entire dataset D, then our problem of finding the (global) top-k-outliers by the following optimization problem as defined in [2]:

$$\underset{O \subseteq D}{min} H(D - O)$$
$$s.t. \ |O| = k.$$



i.e., outlier set is the subset of the entire data getting rid of which from the original dataset minimizes the entropy of the original dataset.
If the variable X is multi-dimensional, then computation of joint entropy involves computation of joint distribution, that can be very expensive, espicially when the number of dimensions are huge. We assume dimensions are independent, we can use **chain rule of entropy** and easily prove the following:
$H(X_1, X_2, \ldots, X_n) = \sum_{i=1}^{n} H(X_i)$,
when X_is are i.i.d.s.

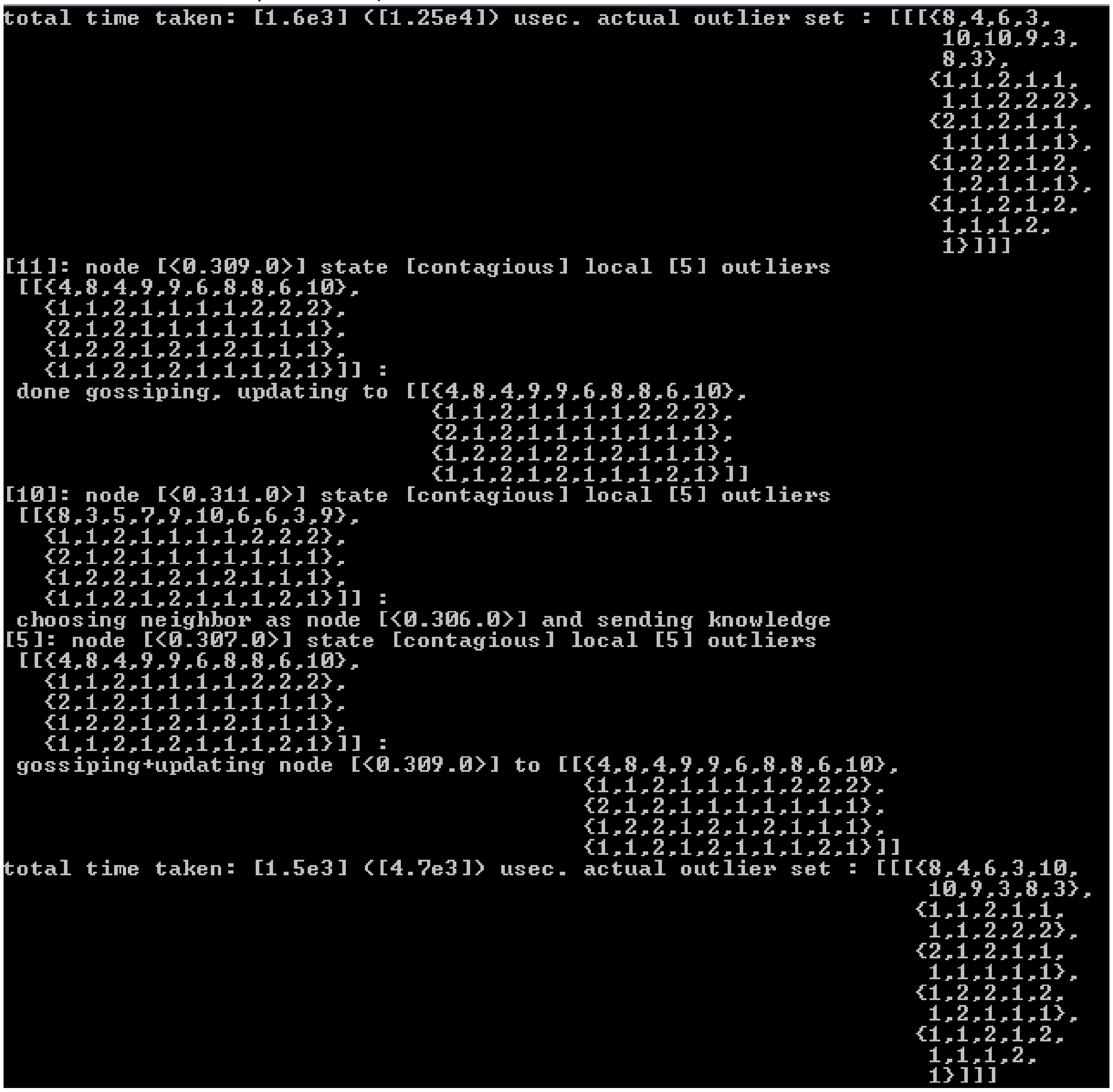
Distributed Extension

In order to extend the problem to its distributed version, we use the above same formulation for finding the local outliers at each and every node. In order to find global outliers from the local outliers stored in each node, we use gossiping with epidemic / rumor spreading. Here we have an important assumption:
A global outlier must appear in at least one of the local outlier set and hence top-k global outlier must be present in the union of all top-k local outlier sets. If O is the global outlier set and O_i are the sets of local outliers, $\forall i = 1 \ldots k \Rightarrow O \subseteq \bigcup_{i=1}^{N} O_i$, where N is the number of nodes in the distributed version of the problem.

Hence once any two nodes find their local top-k outliers, they can gossip and find their global top-k-outliers from their combined local top-k-outliers set.

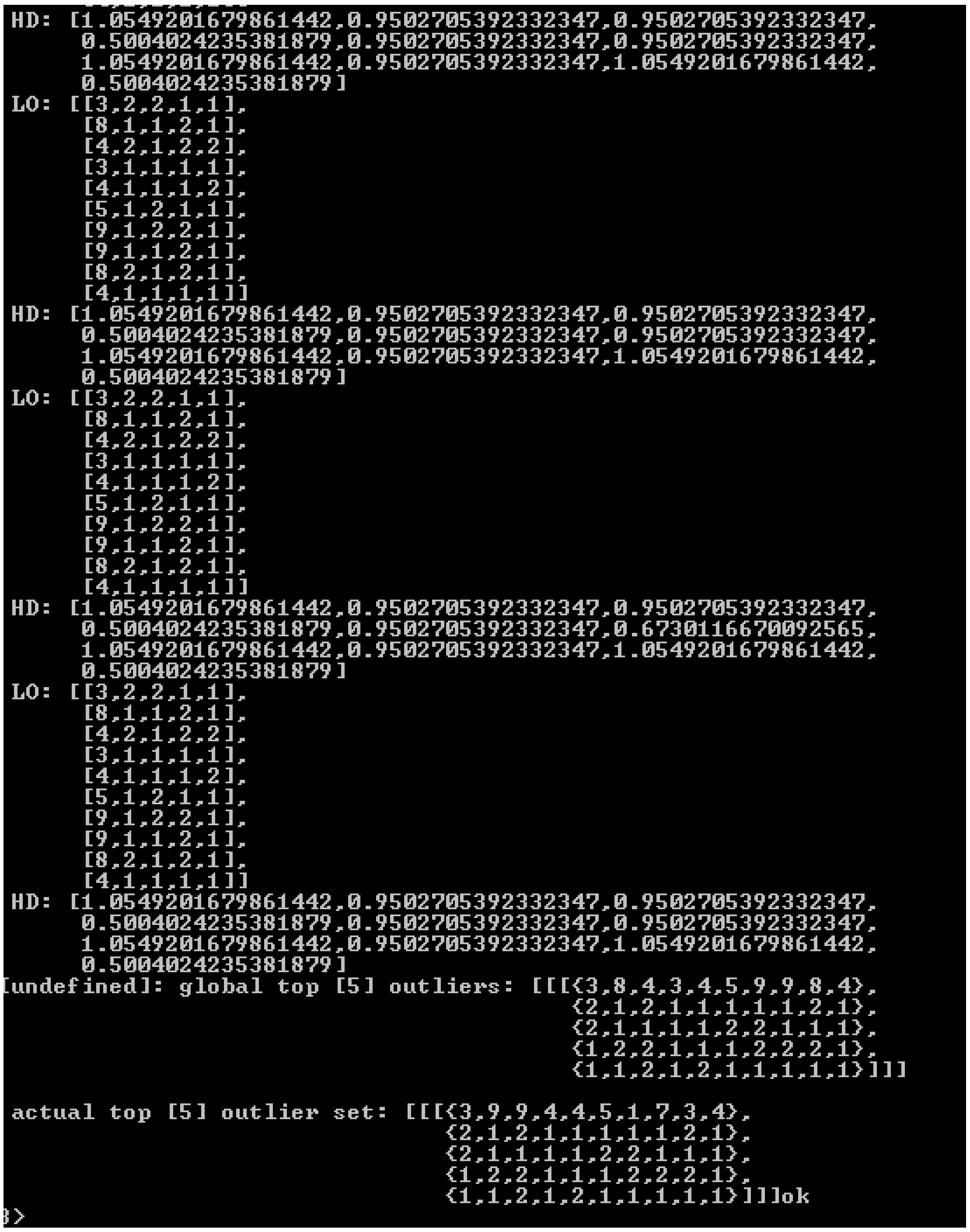
The local outliers are computed and combined to form global outliers via gossip [8] using the same entropy minimization technique. Also the results between distributed and centralized versions are compared using metrics like precision and recall.

Here are some results (run in 10 nodes, each node containing 100 data tuples with 10 dimensions, we are interested to find top-k outliers):



```
total time taken: [1.6e3] ([1.25e4]) usec. actual outlier set : [[[<8,4,6,3,
                                                                    10,10,9,3,
                                                                    8,3>,
                                                                   <1,1,2,1,1,
                                                                    1,1,2,2,2>,
                                                                   <2,1,2,1,1,
                                                                    1,1,1,1,1>,
                                                                   <1,2,2,1,2,
                                                                    1,2,1,1,1>,
                                                                   <1,1,2,1,2,
                                                                    1,1,1,2,
                                                                    1>]]]
[11]: node [<0.309.0>] state [contagious] local [5] outliers
 [[<4,8,4,9,9,6,8,8,6,10>,
   <1,1,2,1,1,1,1,2,2,2>,
   <2,1,2,1,1,1,1,1,1,1>,
   <1,2,2,1,2,1,2,1,1,1>,
   <1,1,2,1,2,1,1,1,2,1>]] :
 done gossiping, updating to [[<4,8,4,9,9,6,8,8,6,10>,
                               <1,1,2,1,1,1,1,2,2,2>,
                               <2,1,2,1,1,1,1,1,1,1>,
                               <1,2,2,1,2,1,2,1,1,1>,
                               <1,1,2,1,2,1,1,1,2,1>]]
[10]: node [<0.311.0>] state [contagious] local [5] outliers
 [[<8,3,5,7,9,10,6,6,3,9>,
   <1,1,2,1,1,1,1,2,2,2>,
   <2,1,2,1,1,1,1,1,1,1>,
   <1,2,2,1,2,1,2,1,1,1>,
   <1,1,2,1,2,1,1,1,2,1>]] :
 choosing neighbor as node [<0.306.0>] and sending knowledge
[5]: node [<0.307.0>] state [contagious] local [5] outliers
 [[<4,8,4,9,9,6,8,8,6,10>,
   <1,1,2,1,1,1,1,2,2,2>,
   <2,1,2,1,1,1,1,1,1,1>,
   <1,2,2,1,2,1,2,1,1,1>,
   <1,1,2,1,2,1,1,1,2,1>]] :
 gossiping+updating node [<0.309.0>] to [[<4,8,4,9,9,6,8,8,6,10>,
                                          <1,1,2,1,1,1,1,2,2,2>,
                                          <2,1,2,1,1,1,1,1,1,1>,
                                          <1,2,2,1,2,1,2,1,1,1>,
                                          <1,1,2,1,2,1,1,1,2,1>]]
total time taken: [1.5e3] ([4.7e3]) usec. actual outlier set : [[[<8,4,6,3,10,
                                                                   10,9,3,8,3>,
                                                                  <1,1,2,1,1,
                                                                   1,1,2,2,2>,
                                                                  <2,1,2,1,1,
                                                                   1,1,1,1,1>,
                                                                  <1,2,2,1,2,
                                                                   1,2,1,1,1>,
                                                                  <1,1,2,1,2,
                                                                   1,1,1,2,
                                                                   1>]]]
```

nodes gossiping to find top-k global outliers from their local-k outliers



```
HD: [1.0549201679861442,0.9502705392332347,0.9502705392332347,
     0.5004024235381879,0.9502705392332347,0.9502705392332347,
     1.0549201679861442,0.9502705392332347,1.0549201679861442,
     0.5004024235381879]
LO: [[3,2,2,1,1],
     [8,1,1,2,1],
     [4,2,1,2,2],
     [3,1,1,1,1],
     [4,1,1,1,2],
     [5,1,2,1,1],
     [9,1,2,2,1],
     [9,1,1,2,1],
     [8,2,1,2,1],
     [4,1,1,1,1]]
HD: [1.0549201679861442,0.9502705392332347,0.9502705392332347,
     0.5004024235381879,0.9502705392332347,0.9502705392332347,
     1.0549201679861442,0.9502705392332347,1.0549201679861442,
     0.5004024235381879]
LO: [[3,2,2,1,1],
     [8,1,1,2,1],
     [4,2,1,2,2],
     [3,1,1,1,1],
     [4,1,1,1,2],
     [5,1,2,1,1],
     [9,1,2,2,1],
     [9,1,1,2,1],
     [8,2,1,2,1],
     [4,1,1,1,1]]
HD: [1.0549201679861442,0.9502705392332347,0.9502705392332347,
     0.5004024235381879,0.9502705392332347,0.6730116670092565,
     1.0549201679861442,0.9502705392332347,1.0549201679861442,
     0.5004024235381879]
LO: [[3,2,2,1,1],
     [8,1,1,2,1],
     [4,2,1,2,2],
     [3,1,1,1,1],
     [4,1,1,1,2],
     [5,1,2,1,1],
     [9,1,2,2,1],
     [9,1,1,2,1],
     [8,2,1,2,1],
     [4,1,1,1,1]]
HD: [1.0549201679861442,0.9502705392332347,0.9502705392332347,
     0.5004024235381879,0.9502705392332347,0.9502705392332347,
     1.0549201679861442,0.9502705392332347,1.0549201679861442,
     0.5004024235381879]
[undefined]: global top [5] outliers: [[[{3,8,4,3,4,5,9,9,8,4},
                                         {2,1,2,1,1,1,1,1,2,1},
                                         {2,1,1,1,1,2,2,1,1,1},
                                         {1,2,2,1,1,1,2,2,2,1},
                                         {1,1,2,1,2,1,1,1,1,1}]]]

 actual top [5] outlier set: [[[{3,9,9,4,4,5,1,7,3,4},
                                {2,1,2,1,1,1,1,1,2,1},
                                {2,1,1,1,1,2,2,1,1,1},
                                {1,2,2,1,1,1,2,2,2,1},
                                {1,1,2,1,2,1,1,1,1,1}]]]ok
3>
```

Here HD denotes entropy of the data set and LO denotes the local outlier set, the nodes gossip and agree upon their local outlier sets and run the same entropy minimization algorithm to update each other's local outlier sets and converge to global outlier sets. As can be seen from the above result, in most of the cases the **precision** (measures what fraction of the outliers computed by the algorithm are actual top-k-outliers or not) and **recall** (measures what fraction of the top-k outliers are actually output) are high (in the above example shown both are 80% though, a bit low, can be increased by increasing number of iterations).

Median of ‘n’ numbers using gossip

Assumptions

The assumptions for initialization of N processes mentioned above for average algorithm also hold good for median algorithm. The number of iterations for the algorithm and the criteria to change the state of process from contagious to infected also remain the same as the average algorithm.

Algorithm:

We have attempted to compute the median of ‘N’ numbers using gossip. The algorithm used is as follows:

1) Initialize ‘N’ processes with some data generated using a uniform random number generator.
2) Make one of the processes to be ‘contagious’, the one that knows the rumor. Other N-1 processes are termed ‘pristine’, that is they do not yet know the rumor.
3) We start with the contagious process initializing the gossip. It selects a neighbor process at random and sends it it’s data.
4) The algorithm then proceeds as follows:
    a. The receiver process on receiving the message compares the data of the sender with it’s own data.
    b. If the data of the sender is greater than it’s own data, then add the sender processes’ id to ‘GreaterThan’ list maintained with the receiver process.(GreaterThan is a list of all process ids whose data is greater than the receivers data)
    c. If the data of the sender is less than or equal to the data of the receiver, add the process id of the sender process to the LessThan list maintained with the receiver process (LessThan is a list of all process ids whose data is less than or equal to the receivers data.)
    d. After this comparison, it sends an ‘update’ message back to the sender process so as to update the sender’s LessThan and GreaterThan lists. The sender then follows the same process mentioned above in points a,b and c to update the lists.
    e. During the above procedure if the process notices that the length of it’s LessThan and GreaterThan lists is
        i. each N/2(for N=odd) : That process prints that it is the median
        ii. either (N/2 and N/2-1 respectively) or (N/2-1 and N/2 respectively) (for N=even) : The process sends back a message to the main parent process saying that it is in the middle of all processes. The main process waits for two such processes to contact it.

       Else it just picks up a random neighbor and continues the gossip.
    f. For N=even, the main process is contacted by two middle processes saying that they are in the middle. The main process averages their data and prints as the median.
    g. The above process iterates till the median is located or the iterations time out.

The above algorithm requires at least N*N*log(N) iterations before it can locate the median. This is because each process has to talk to other N-1 processes at least once before it can decide where it stands in the entire list of processes. Also, the algorithm works well with odd values of N that even values.

References

[1] Joel W. Branch, Boleslaw K. Szymanski, Chris Giannella, Ran Wolff, and Hillol Kargupta. In-network outlier detection in wireless sensor networks. In *ICDCS*, page 51, 2006.

[2] Zengyou He, Xiaofei Xu, Shengchun Deng. **An Optimization Model for Outlier Detection in Categorical Data**

[3] Inderjit S. Dhillon. Co-clustering documents and words using bipartite spectral graph partitioning. In *KDD*, pages 269–274, 2001.

[4] Haimonti Dutta, Chris Giannella, Kirk D. Borne, and Hillol Kargupta. Distributed top-k outlier detection from astronomy catalogs using the demac system. In *SDM*, 2007.

[5] Edwin M. Knorr, Raymond T. Ng, and V. Tucakov. Distance-based outliers: Algorithms and applications. *VLDB J.*, 8(3-4):237–253, 2000.

[6] Ying Liu and Alan P. Sprague. Outlier detection and evaluation by network flow. *IJCAT*, 33(2/3):237–246, 2008.

[7] Spiros Papadimitriou, Hiroyuki Kitagawa, Phillip B. Gibbons, and Christos Faloutsos. Loci: Fast outlier detection using the local correlation integral. In *ICDE*, pages 315–, 2003.

[8] Anne-Marie Kermarrec Maarten van Steen. **Gossiping in Distributed Systems**