DCS Project Report

Reported By: - Piyush Chandra

Part A

Abstract:

In this project we must identify how fast the EVC grows as a function of number of events executed by the process. Once it executes 32n bits we must check how many events it took for the EVC to grow and after it reaches 32n bits, we can reset the system. Same steps should be repeated for 64n bits.

Introduction:

A vector clock does not perform well to track the causality relationship in a large distributed system since it needs to maintain a vector of size n, where n is the number of processes which can be very large. Encoded Vector Clock(EVC) is the solution provided to tackle the size of the vector clock using prime numbers.

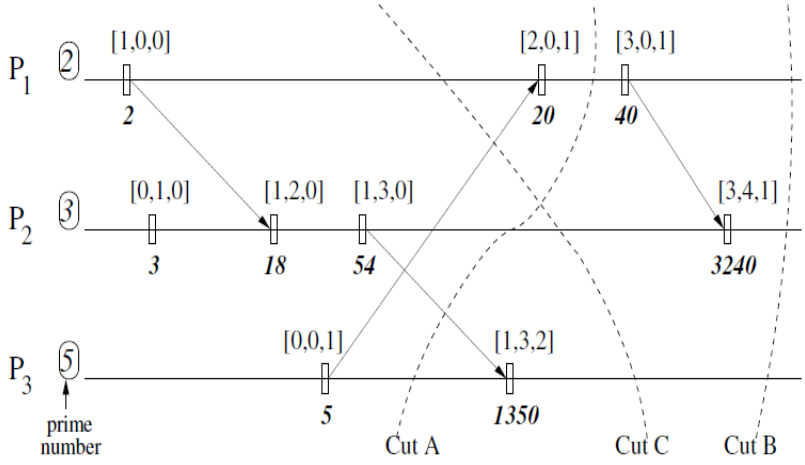


Figure 1: Illustration of using EVC. The vector timestamps and EVC timestamps are shown above and below each timeline, respectively. In real scenarios, only the EVC is stored and transmitted.

Approach:

Simulator Design

The vital part of this project was to create an asynchronous system(simulator) in which as shown in Figure 1 there are local ticks and send events are occurring at a certain probability. We also must make sure that for each send event there is a receive event at some other process. This system has been designed using multithreading.

Initially, the system asks the user how many processes he wants to create for the system. As per the input the system creates that number of threads and assigns each thread a prime number. Then system asks how many events he wants to create for each process and enter the probability of internal event. Next it asks is it testing for 32n bits or 64n bits. For our scenario to occur user should enter a lot of events so that it crosses the 32n and 64n bits. Now as soon as all inputs are entered, the system starts creating an asynchronous system.

In this system, initially as per the probability of internal event, system generates those number of internal events and the rest are made as send events. Each event at each process are inserted in a priority queue which is created for each process. The events are assigned an event id randomly based on which comparator assigns them the position in each priority queue. I make sure that the events ids range is large enough to accommodate the receive events. Also, while creating send events, all these events were given a destination process id which were randomly generated.

Next, we iterate each process priority queue to find the send events and for each send events we make sure at its destination we create a receive event. These receive events were given a unique event id randomly. One thing to notice is that in the simulation I am not allowing receive event to be the first event of any process just to make the simulation smooth, and it does not have any impact on the simulation. Now the system is ready with every process having a priority queue with send, receive and internal events. We just need to dequeue the events from the priority queue to simulate events in asynchronous environment.

Finally, the system starts the threads. The process starts dequeuing its events from its priority queue. Now a problem may occur while dequeuing if a process gets a receive event before its send occurs from its source. Because of this the system may have to wait infinitely for its send and other process which had to receive from this process may also wait for it to send. This may be a deadlock state. This state the system will tackle by pushing the receive event for which the sender didn't send any send event at the end of its priority queue by changing event id to its maximum event id + 1. This makes sure that the system does not reach deadlock condition.

EVC Processing

Whenever an event occurs at a process, the system updates its EVC according to the below rules:

1) Initialize $t_i = 1$.
2) Before an internal event happens at process P_i, $t_i = t_i * p_i$ (local tick).
3) Before process Pi sends a message, it first executes $t_i = t_i * p_i$ (local tick), then it sends the message piggybacked with t_i.
4) When process P_i receives a message piggybacked with timestamp s, it executes
   $t_i = LCM(s, t_i)$ (merge);
   $t_i = t_i * p_i$ (local tick) before delivering the message.

To store the EVC, I have used a global map which contains process number as its key and the process latest EVC as the value. For the local tick, I can get the EVC from this global map. Also, I am maintaining a separate map to store the EVC map of sender process number and its EVC value which I use when a receive event occurs at the destination to find EVC value that has been sent by the sender. Therefore, whenever a receive event occurs, I find in this map if any sender has sent to this process and if it has then I use that EVC value to find LCM as shown in step 4 in the table above. For finding LCM, BigInteger class has its own method for GCD calculation using which I calculated LCM. Finally, when EVC size crosses 32n or 64n we do system exit.

Introduction of BigInteger

The EVC grows so fast because of the calculations shown in above table that the Integer or Long datatype fails miserably for this experiment. So, I had to take a datatype which can store big values. This problem was solved using BigInteger of Java. BigInteger stores values as int array. So, you can store as many values in this and the array size will keep on increasing unboundedly.

Result:

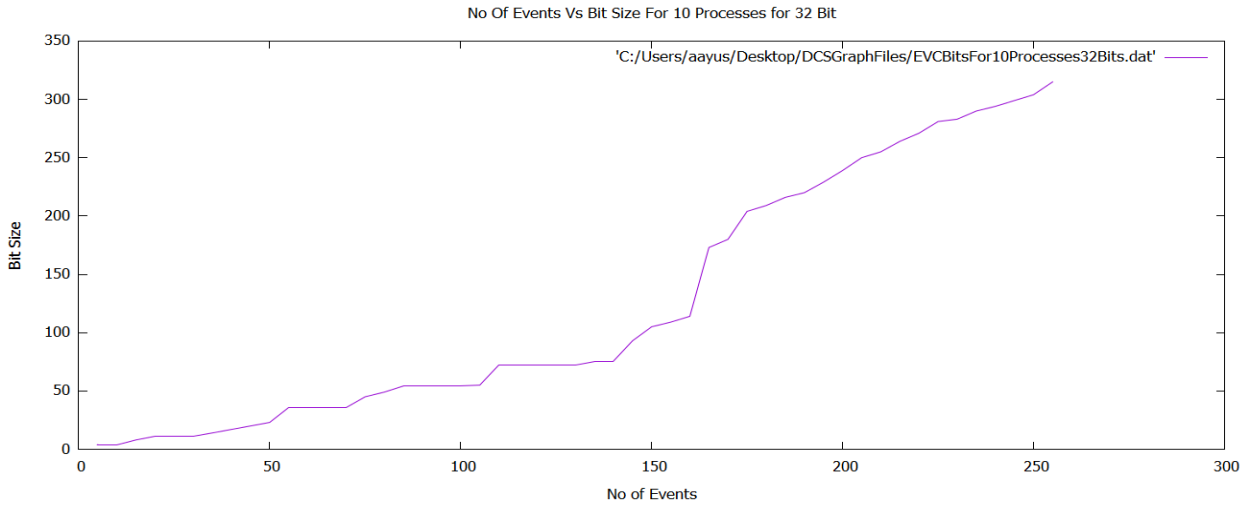
The events have been executed until the EVC crosses the 32n size where n is the number of processes.

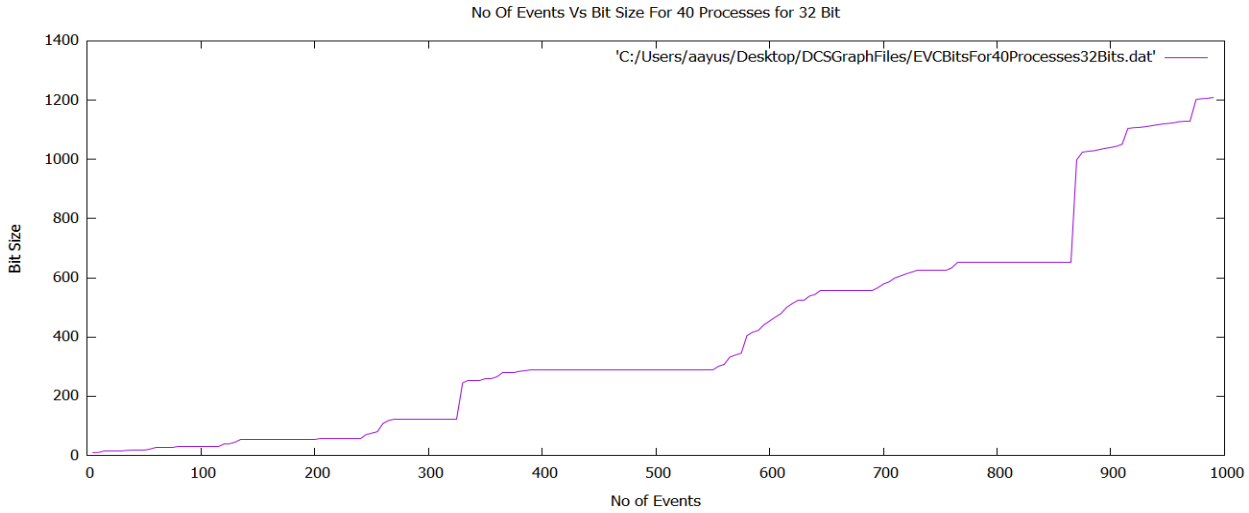
Number of Events Versus Bit Size

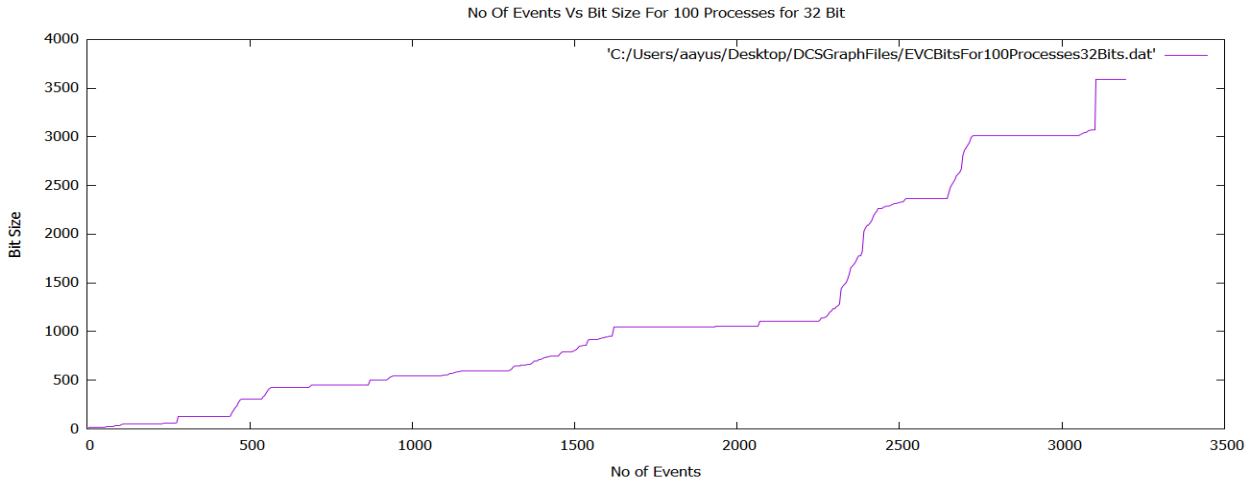
For plotting this graph, I have found maximum EVC size of all the processes at intervals of 5 and have recorded them after taking average of 3 runs.

Next, I have plotted number of events versus its bit size until the bit size crosses 32n for 10, 40 and 100 n(processes).

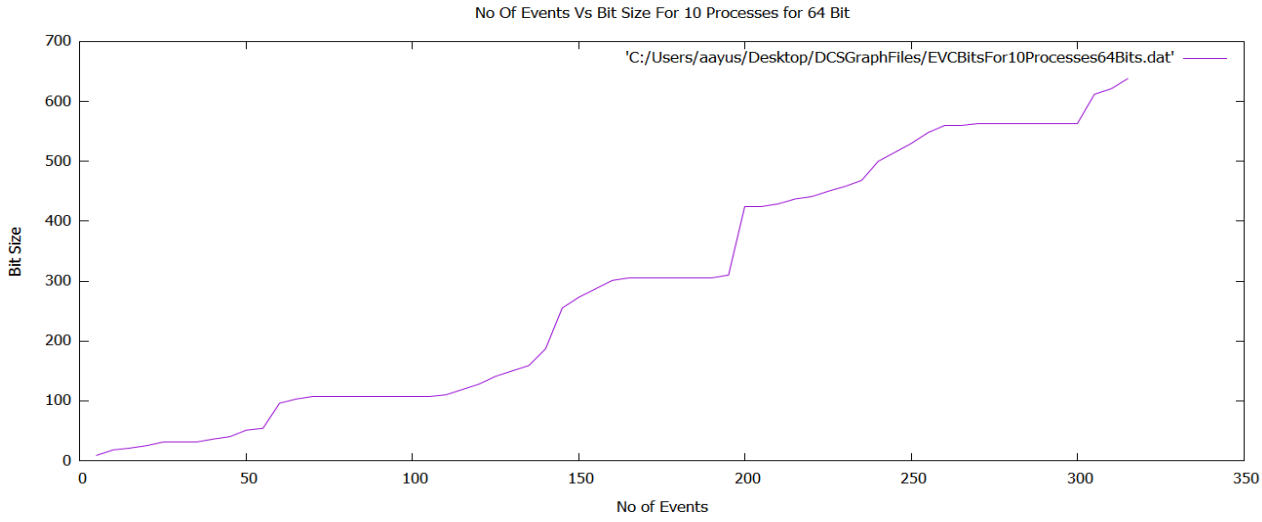
Note: For all the computations in this project, I have taken internal events as 40 percent of total events.

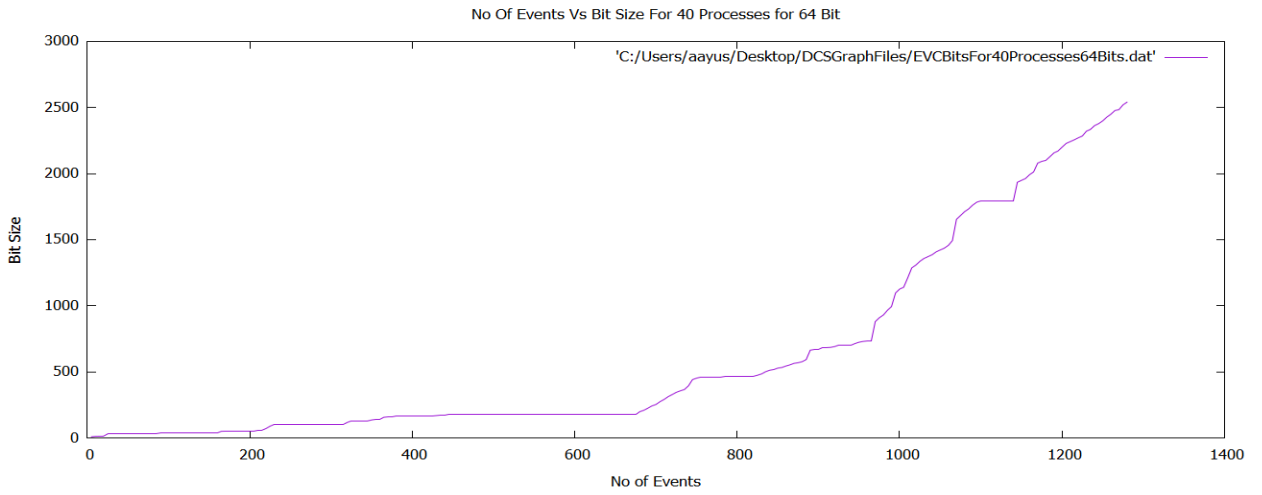


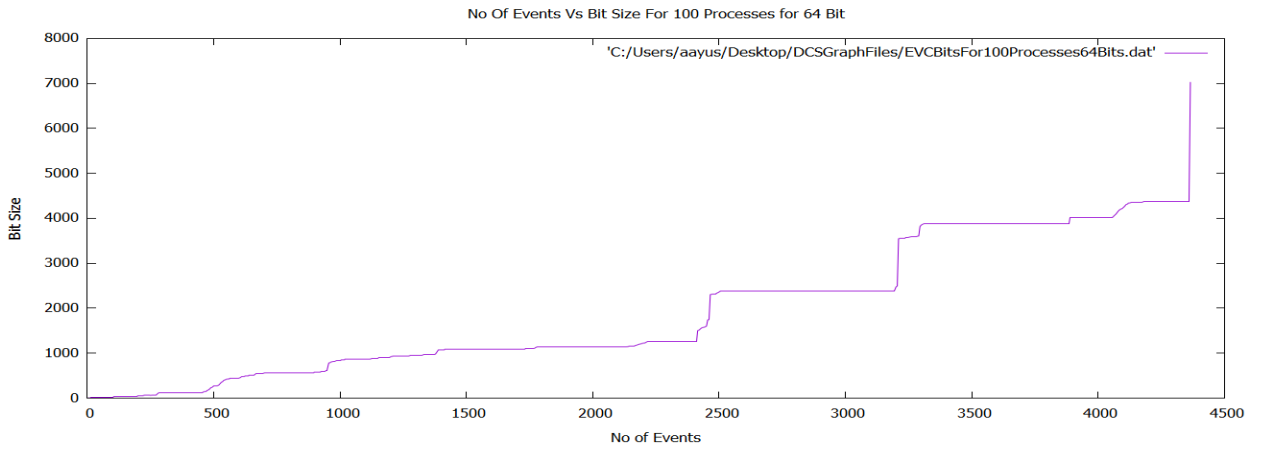




The below graphs are of 3 recordings for 64n bits.



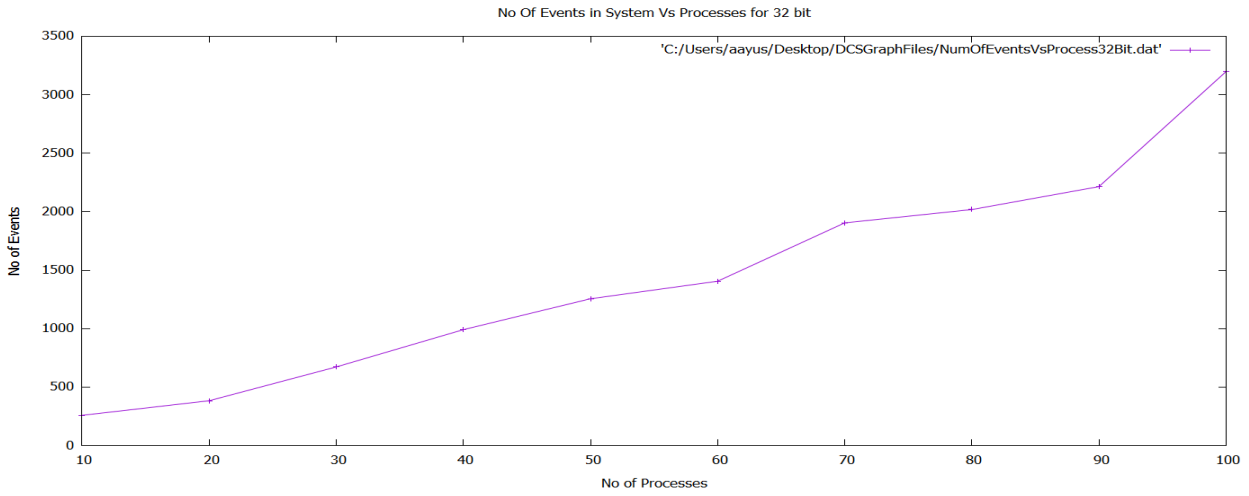


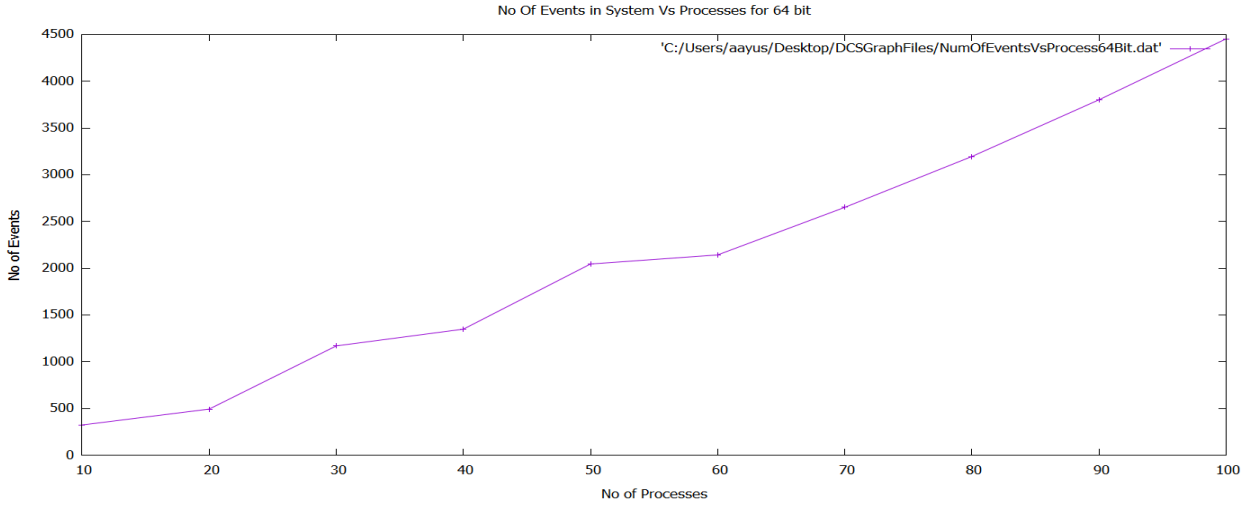


Inference: As expected the size of EVC grows as the number of events increases. Also, the number of events is increasing for 64 bits since more number of events are executed. At times we see a sharp rise in graph and then it becomes smooth because the EVC grows significantly fast.

Number of Processes Vs Number of Events Until EVC Reaches 32n

Now I have also taken recording of total number of events as the number of processes increases till the EVC reaches 32n bits. For this, I updated count whenever any event occurred and finally taken average at each. Therefore for 10 processes how many events occurred, for 20 how many events occurred and so on till 100.





Inference: We see that a curve like graph is occurring as the number of events are occurring more as the number of processes are increasing. Thus, as the number of processes grow the number of events grow more steeply.

Part B

Introduction:

The EVC grows very fast, because of which overflow is inevitable, so we need to find a solution to store EVC which can take less space.

Abstract:

We use logarithm to store the EVC. But since logarithm involve finite-precision arithmetic, we will get errors because of the limited precision. So, we need to analyze and be careful with the usage of precision, logarithm base, and rounding of values. We must find the percentage of errors introduced due to the limited precision.

Approach:

Arithmetic Precision Library

In this experiment we will be getting values in decimals, so BigInteger cannot be used here. I tried with BigDecimal but I was struggling to find logarithms and anti-logarithms since it was not supporting them. Finally, I decided to go with ApFloat library since it has all the required functions what we need for this project and has a good documentation.

EVC Processing

Now because of the introduction of logarithm the system will update the EVC according to the following:

1) Initialize $t_i = 0$.
2) Before an internal event happens at process P_i,
   $t_i = t_i + \log(pi)$ (local tick).
3) Before process P_i sends a message, it first executes
   $t_i = t_i + \log(pi)$ (local tick), then it sends the message piggybacked with t_i
4) When process P_i receives a message piggybacked with timestamp s, it executes
   $t_i = s + t_i - \log(GCD(\log-1(s), \log -1(t_i)))$ (merge);
   $t_i = t_i + \log(p_i)$ (local tick)
   before delivering the message.

Here, during GCD computation, I have rounded the EVC using ceil function of Apfloat which rounds to the next integer if decimal is present. This gave me the best result. The simulation used was like the one used in Part A except that now simulator runs until number of events equal to v*n where v=50 and n is number of processes. New EVC computation was done using the table above. Then, I stored the original vector clock as the key of map and new EVC as the value of the map.

Finally, I compared all the pairs generated by the system, and classified it as False Positive if according to original vector clock, there is a causality relationship and 1st element happened before 2nd, but new EVC gives 2nd element happened before 1st (1st occurred causally before 2nd). This I did by converting the Apfloat EVC to double to reduce its precision to 53 bits. So, if mantissa contains zeroes till 53 bits, it doesn't matter which number is after 53 bits and that will be considered as a natural number. This decreases error and improves accuracy. Similarly, if 1st element's original EVC is less but the new EVC gives opposite, than it is classified as False Positive.

And the formula used for calculating False Positive and False Negative error rate is:

FP error rate % = (FP/(FP+TN)) * 100

FN error rate % = (FN/(FN+TP)) * 100

Result:

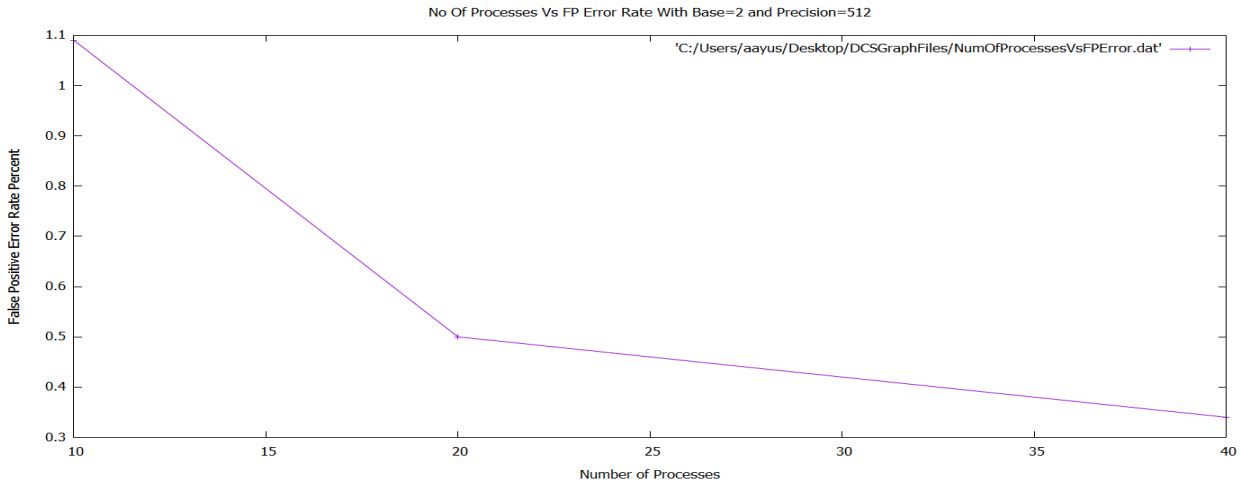
The results for different precisions are reported in each table below for v=50:

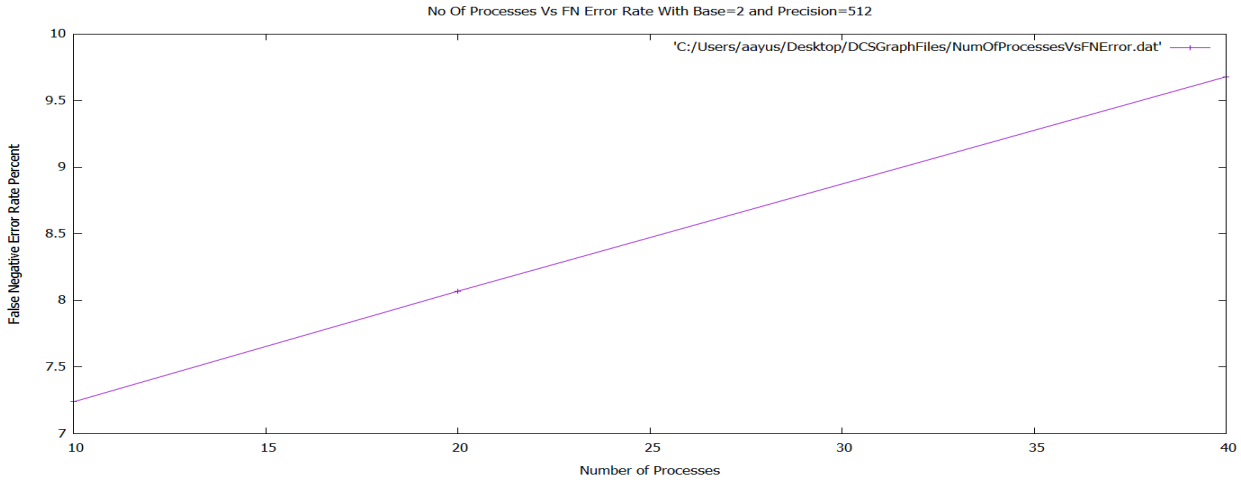
No of Process	Log Base	Precision Bits	Total Events	Total Events Pairs	False Negative	False Positive	False Negative Percent	False Positive Percent
10	2	64(20)	500	124750	1412	534	5.62	0.90
20	2	64(20)	1000	499500	2316	5273	7.41	2.24
40	2	64(20)	2000	1999000	5295	13733	7.96	1.44
10	10	64(20)	500	124750	2126	998	7.29	1.76
20	10	64(20)	1000	499500	2397	4163	7.84	1.80
40	10	64(20)	2000	1999000	4663	11549	7.08	1.23
10	25	64(20)	500	124750	1409	1091	6.84	1.95
20	25	64(20)	1000	499500	2680	2665	7.04	1.12
40	25	64(20)	2000	1999000	5152	12818	6.86	1.36

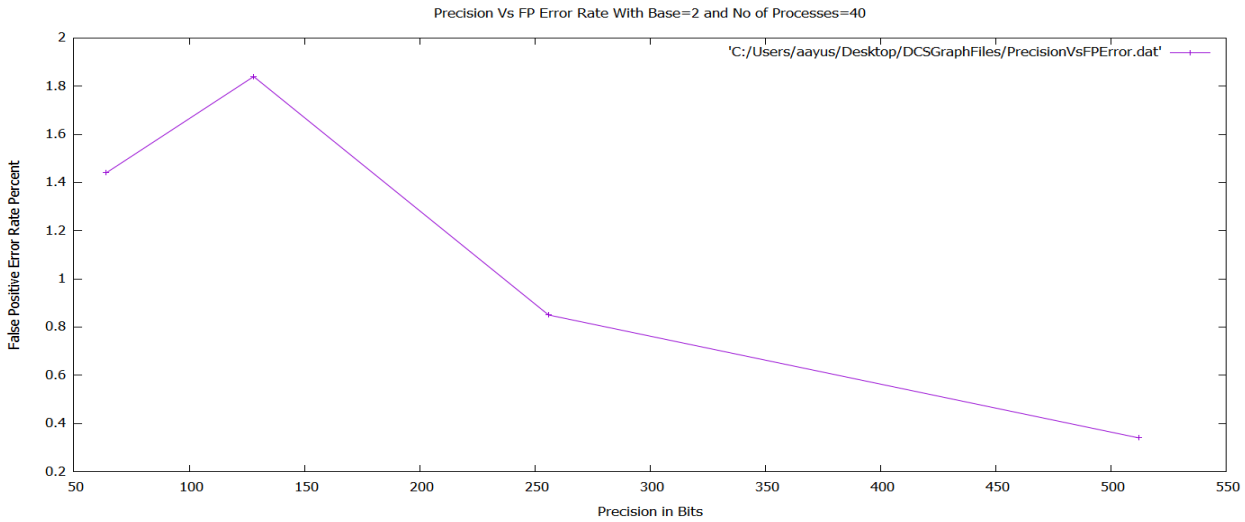
No of Process	Log Base	Precision Bits(Length)	Total Events	Total Events Pairs	False Negative	False Positive	False Negative Percent	False Positive Percent
10	2	128(39)	500	124750	943	707	7.55	1.15
20	2	128(39)	1000	499500	2562	7818	6.08	3.16
40	2	128(39)	2000	1999000	4509	16954	8.06	1.84
10	10	128(39)	500	124750	1408	538	11.87	0.90
20	10	128(39)	1000	499500	2499	1187	8.77	0.51
40	10	128(39)	2000	1999000	4545	10660	7.28	1.11
10	25	128(39)	500	124750	1019	947	6.50	1.67
20	25	128(39)	1000	499500	2101	2864	7.08	1.24
40	25	128(39)	2000	1999000	4296	14363	7.82	1.55

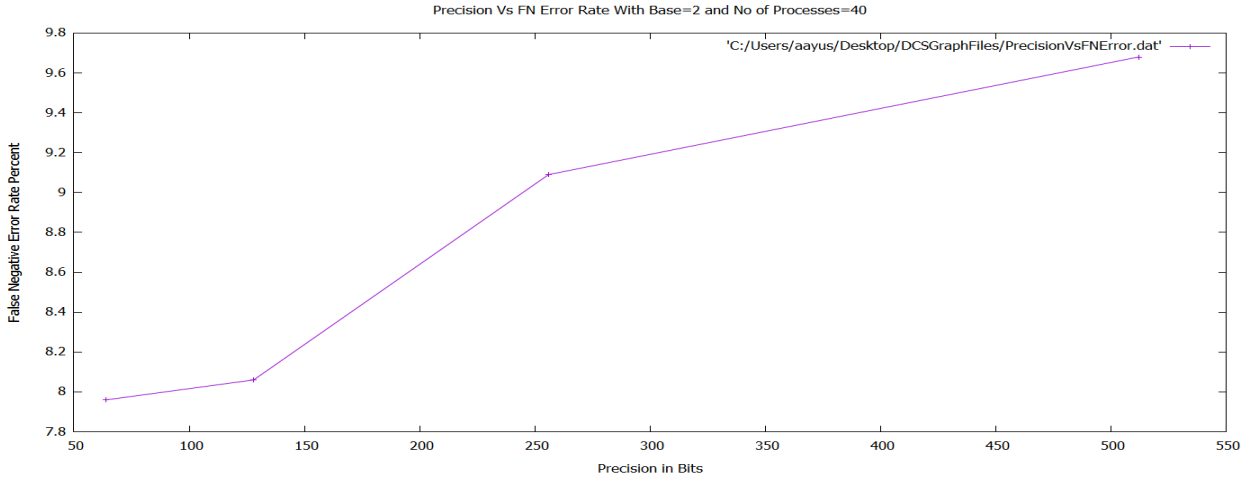
No of Process	Log Base	Precision Bits(Length)	Total Events	Total Events Pairs	False Negative	False Positive	False Negative Percent	False Positive Percent
10	2	256(78)	500	124750	1358	415	6.47	0.76
20	2	256(78)	1000	499500	2543	4309	7.59	1.86
40	2	256(78)	2000	1999000	4575	7834	9.09	0.85
10	10	256(78)	500	124750	1854	825	9.53	1.46
20	10	256(78)	1000	499500	2832	6886	10.73	2.97
40	10	256(78)	2000	1999000	5065	7728	8.86	0.82
10	25	256(78)	500	124750	1503	1126	7.58	2.09
20	25	256(78)	1000	499500	2503	4205	10.57	1.80
40	25	256(78)	2000	1999000	4559	9765	9.12	1.08

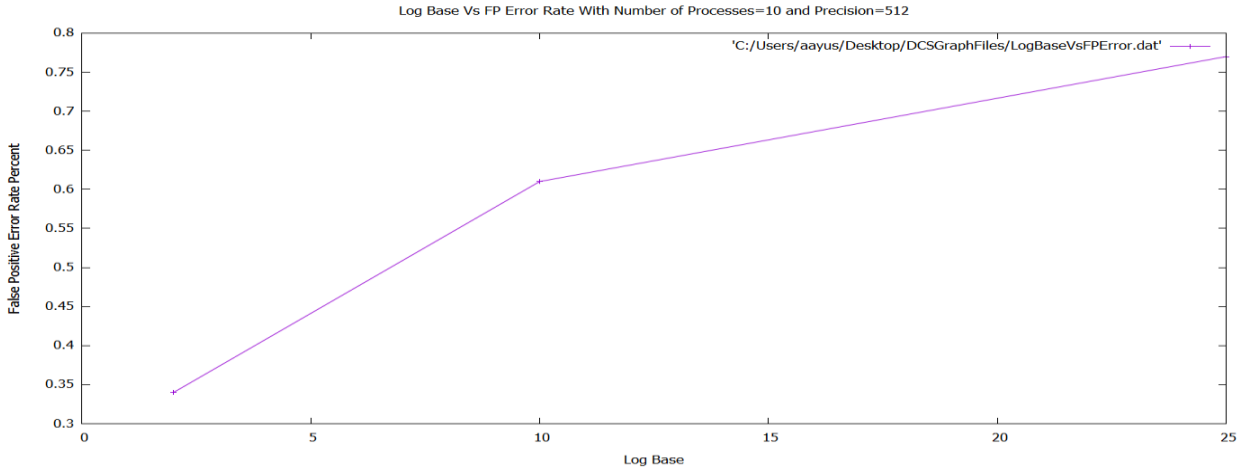
No of Process	Log Base	Precision Bits(Length)	Total Events	Total Events Pairs	False Negative	False Positive	False Negative Percent	False Positive Percent
10	2	512(155)	500	124750	1273	618	7.24	1.09
20	2	512(155)	1000	499500	2039	1171	8.07	0.50
40	2	512(155)	2000	1999000	4584	3229	9.68	0.34
10	10	512(155)	500	124750	15887	58389	9.46	0.79
20	10	512(155)	1000	499500	2084	3568	9.90	1.52
40	10	512(155)	2000	1999000	4258	5729	8.72	0.61
10	25	512(155)	500	124750	1426	1094	7.95	1.85
20	25	512(155)	1000	499500	2121	1419	6.49	0.63
40	25	512(155)	2000	1999000	4488	7366	8.93	0.77

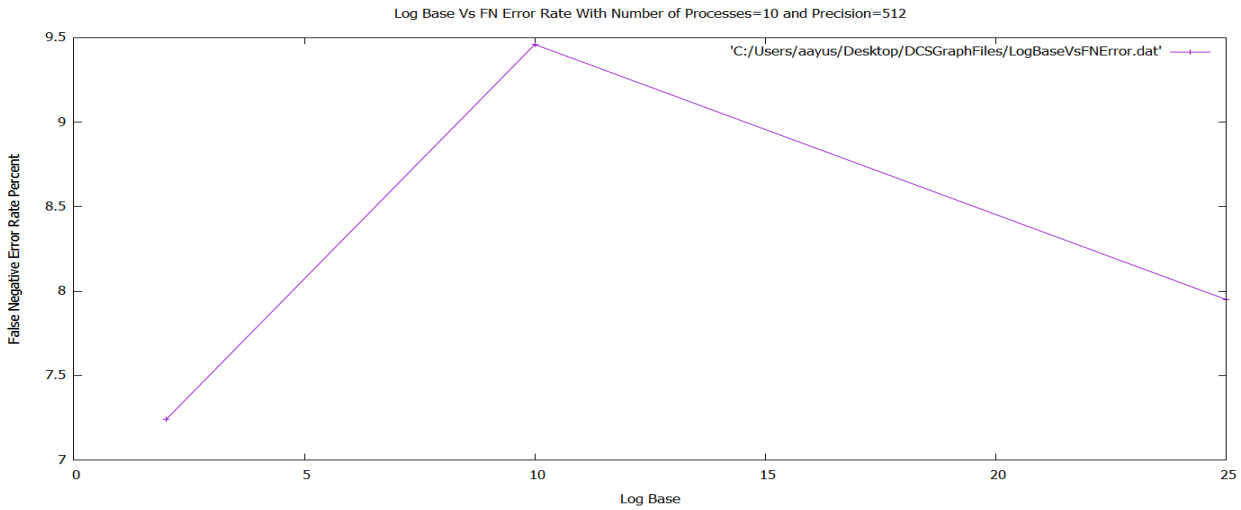

No Of Processes Vs FP Error Rate With Base=2 and Precision=512
'C:/Users/aayus/Desktop/DCSGraphFiles/NumOfProcessesVsFPError.dat'
False Positive Error Rate Percent
Number of Processes
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
10
15
20
25
30
35
40


No Of Processes Vs FN Error Rate With Base=2 and Precision=512
'C:/Users/aayus/Desktop/DCSGraphFiles/NumOfProcessesVsFNError.dat'
False Negative Error Rate Percent
Number of Processes
10
9.5
9
8.5
8
7.5
7
10
15
20
25
30
35
40


Precision Vs FP Error Rate With Base=2 and No of Processes=40
'C:/Users/aayus/Desktop/DCSGraphFiles/PrecisionVsFPError.dat'
False Positive Error Rate Percent
Precision in Bits
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
50
100
150
200
250
300
350
400
450
500
550


Precision Vs FN Error Rate With Base=2 and No of Processes=40
'C:/Users/aayus/Desktop/DCSGraphFiles/PrecisionVsFNError.dat'
False Negative Error Rate Percent
Precision in Bits
9.8
9.6
9.4
9.2
9
8.8
8.6
8.4
8.2
8
7.8
50
100
150
200
250
300
350
400
450
500
550


Log Base Vs FP Error Rate With Number of Processes=10 and Precision=512
'C:/Users/aayus/Desktop/DCSGraphFiles/LogBaseVsFPError.dat'
False Positive Error Rate Percent
Log Base
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0
5
10
15
20
25



Inference: As we observe, as the number of processes increases the false negative error percent increases, but this is opposite for False Positive Error Rate. Also, we see that for most of the graphs high base. Increasing the precision gives us reduction in false positive error rate but increase in error for false positive. Higher base is giving high error for false positive but initial increase and then decrease in false negative error rate.

References:

1. Research Paper - Ajay D. Kshemkalyani, Ashfaq A. Khokhar, Min Shen: Encoded Vector Clock: Using Primes to Characterize Causality in Distributed Systems https://dl.acm.org/citation.cfm?doid=3154273.3154305
2. Project Description https://www.cs.uic.edu/~ajayk/EVCGrowthRate.pdf
3. Arithmetic Precision Details https://en.wikipedia.org/wiki/Arbitrary-precision_arithmetic
4. Library available for Arithmetic Precision https://en.wikipedia.org/wiki/List_of_arbitrary-precision_arithmetic_software
5. Apfloat Documentation http://www.apfloat.org/apfloat_java
6. Project Implementation on GitHub https://github.com/piyush910/Personal-Projects/tree/master/Encoded%20Vector%20Clock