Distributed Computing Systems

EVC Growth Rate Simulation

Arturo Cardone – arturocardone35@gmail.com

- **Implementation:**

The simulation was written in Java programming language using a Round Robin execution strategy. For this reason, a single thread was used in order to simplify the program. The execution of the simulation works as follows: first of all, the set of processes that will take part in the experiment is initialized with an unique prime number each, generated through the Sieve of Eratosthenes algorithm. Then, the main loop commences, consisting of a sequence of iterations that continues until a maximum number of events or EVC overflow are reached.

At each new iteration (which corresponds to a system timestamp) of the system, a new event is generated (internal or send according to a percentage value indicated as an input parameter) and added to the priority queue of the process that is currently being served in the round-robin strategy. The generated event is not added to the queue for the current timestamp only in the case in which a receive operation is scheduled to be executed in this specific timestamp. In this way, it is not possible to delay receive events in time. The current process then starts its execution by polling a new event from its priority queue (which sorts the events according to their 'iteration timestamp') and executes it accordingly. If it happens to be a send event, then the corresponding receive event is inserted in the queue of another randomly selected process, simulating a delay of [0-10] time units chosen in a random way.

In order to be able to execute all kinds of tests, the simulation manages Vector Clocks, EVCs and logarithmic representations of EVCs by storing all of them in each process. For this purpose, local ticks and merge operations are executed for all of the three kinds of clocks during each event that necessitate them.

Each process is represented as a Java object containing a priority queue of events to be executed (events are objects as well, containing piggybacked EVC, Vector Clock, or logarithm of EVC), its assigned prime number, and the set of current EVC, Vector Clock, and EVC log.

In order to manage the EVC numbers, standard Java types were not sufficient, so the *BigInteger* type was adopted to have potentially unbounded bit storage for those numbers. This Java type simply stores integers as arrays of digits. The same goes for decimal numbers, as for a safe storing of EVC logarithms, the *BigDecimal* type was used.

To be able to calculate the GCD, the Euclidean recursive algorithm has been adopted.

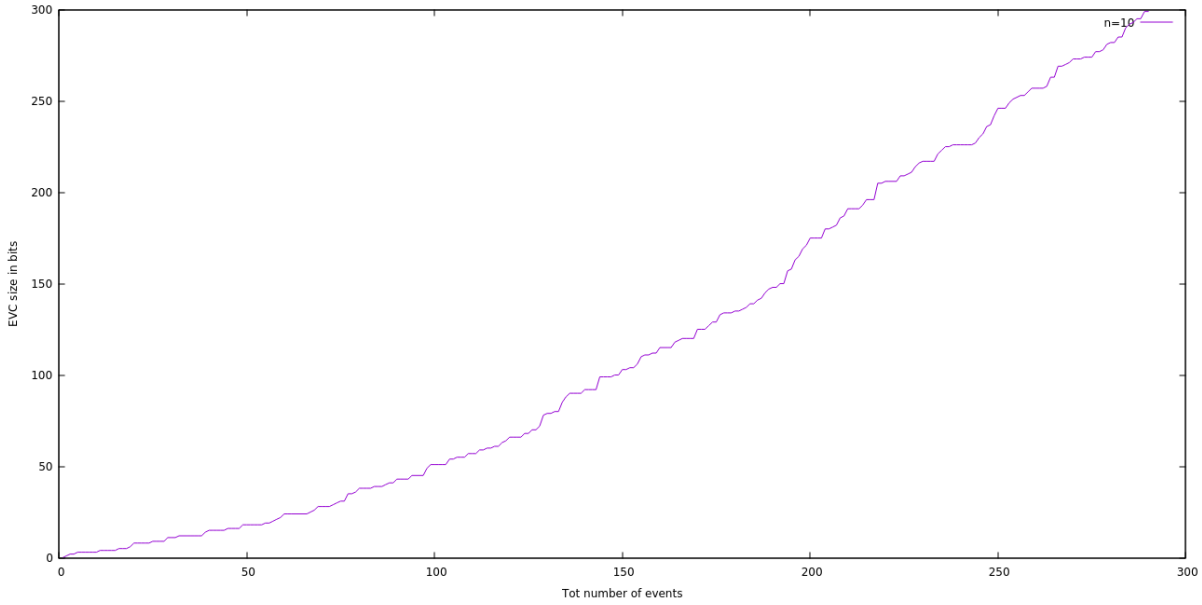
After each iteration, the executed events are stored in an 'History' list that is then used by a final analytic function, which composes each possible pair of executed events in order to analyze the causal dependency conditions necessary for the second part of the experiment.

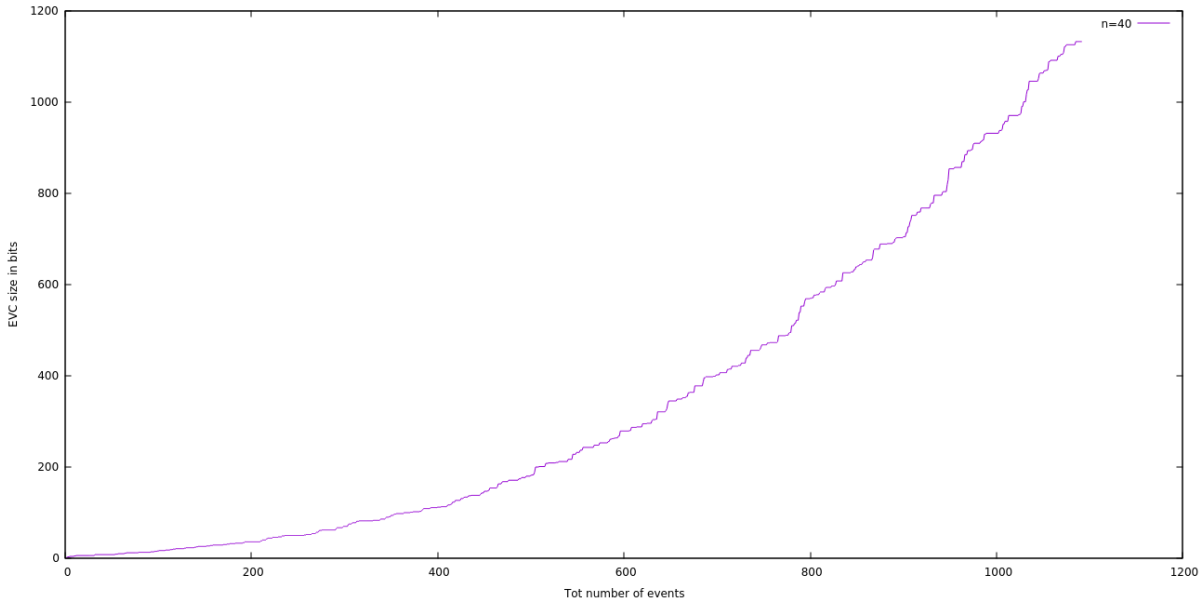
- **Part 1:**

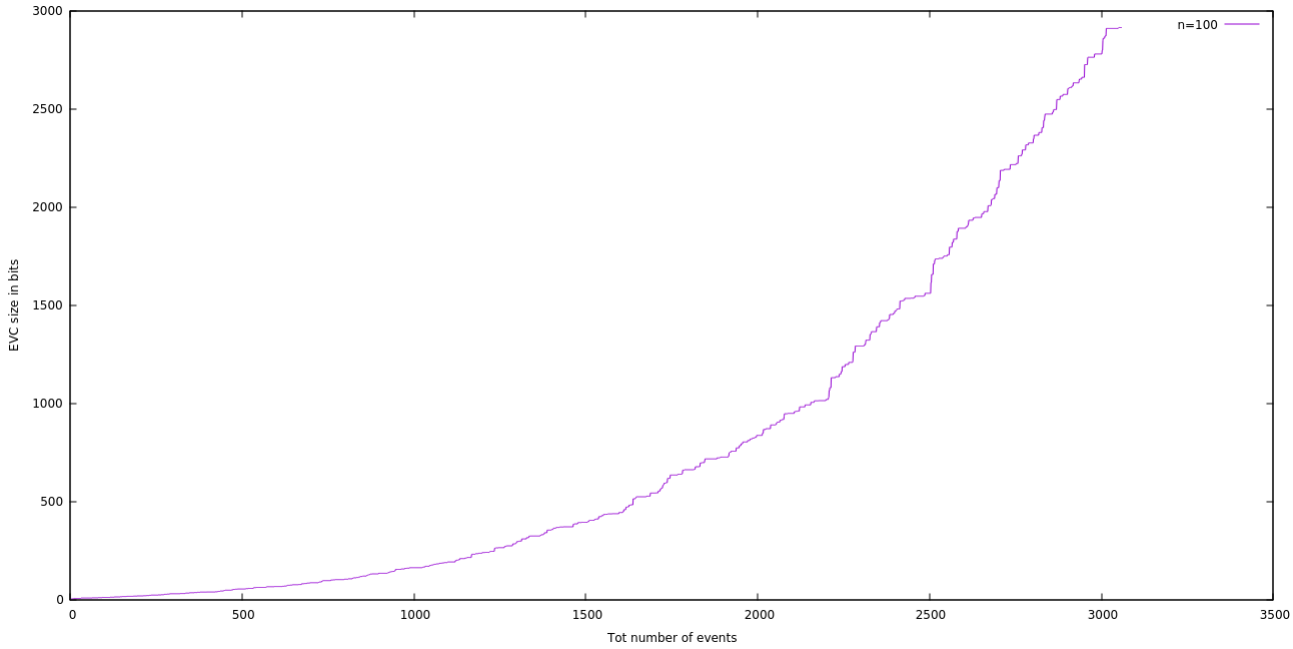
In the first part of the experiment, the goal was to understand the behavior of the EVC growth. The program had two main input values:

- The number of processes that take part in the simulation. This obviously influences the number of events executed in the system and the number of messages sent among the processes themselves.
- The percentage of internal events in respect to send and receive type of events: this mainly influences the number of messages sent in the system and, consequentially, the number of "merge" actions executed, which are able to speed up the EVC growth.

The first experiment consisted in analyzing the EVC size in bits in respect to the total number of events executed in the system. In order to gather all the needed data, the simulation was run until EVC overflow happened at the earliest process. The experiment was executed for a number of processes equal to 10, 40 and 100:





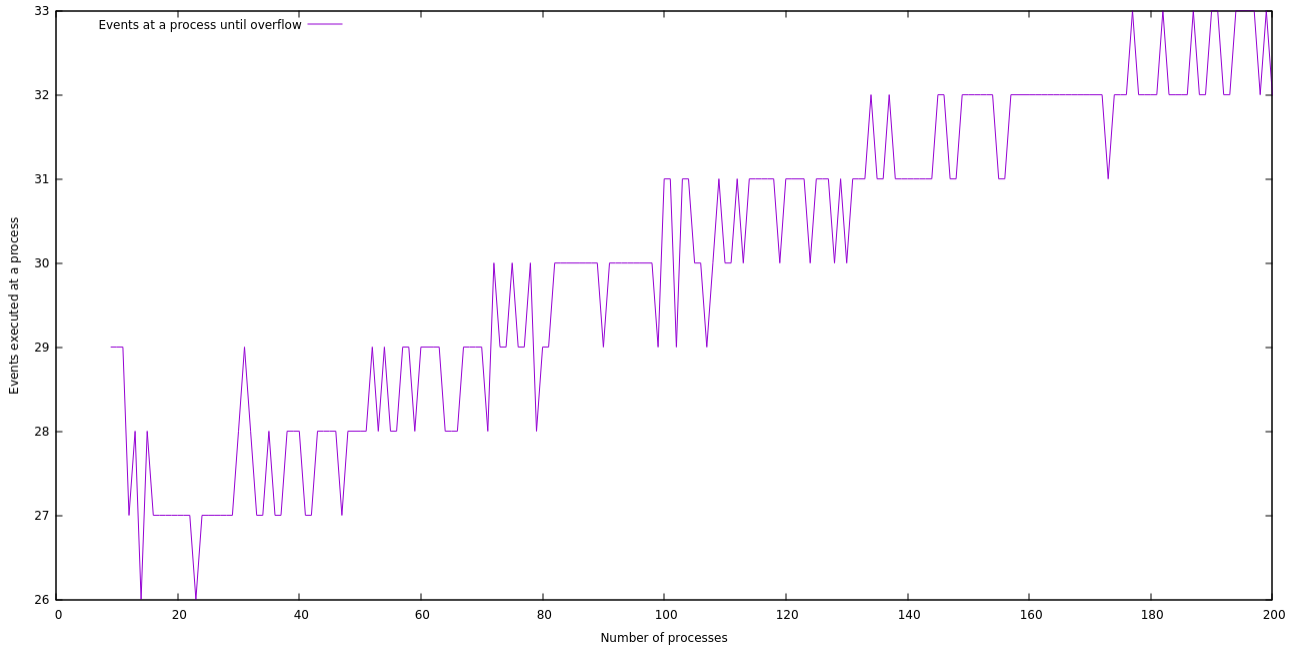


All three charts are referred to a 50% of chance of internal events in respect to send events. Also, in order to obtain reliable results, 10 runs were executed for each case (n=10, 40, 100), and the average value of EVC size in bits for each number of events was selected to be represented on those charts.

It is easy to notice how in all three cases the size of the EVC in bits increases exponentially with the total number of events that have been executed in the system. This effect is way more noticeable when an higher number of processes is selected: this is because the size of the EVC will increase with a speed that is always greater, considering that we keep multiplying an increasingly big number representing the clock with the prime number corresponding to a specific process. Also, an higher number of processes implies that some of those will have a very big prime number associated to them (which will be multiplied by the EVC value): this speeds up the EVC growth even more rapidly.

Other than that, we can observe how in all three of those cases, each process executes about 29-30 events before EVC overflow is reached in the system. This can be explained by considering that the EVC overflow value (32*n) becomes greater with a greater number of processes. Greater is also the EVC growth rate caused by an increasingly big prime number. Evidently, those two grow at a similar pace, granting a pretty constant number of events to be executed at each process, independently of the total number of processes in the system.

The second experiment consisted of analyzing how the number of events executed at a single process changes in respect to the number of processes *n.* This could be done simply by starting the execution of the simulation using a number increasing in the range [10, 200] as the parameter for the number of processes participating in the execution. Then, the total number of internal, send and receive events is summed up for each process. Finally, in order to reduce the effect of outliers, the average number of events executed at a process among all of the processes has been used in the calculations.

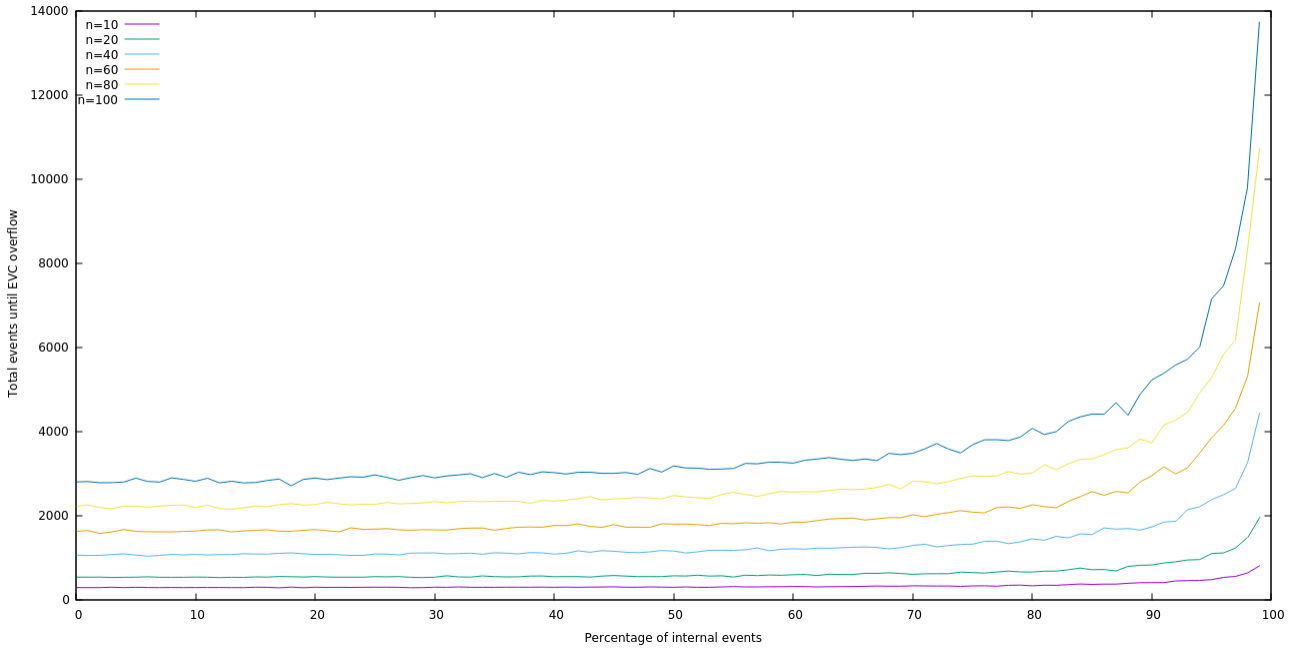


The data represented in this chart has been gathered as the average set of values of 10 runs of the simulation. During all of those, there was 50% chance of internal events in respect to send events. For each value of number of processes *n*, the execution was stopped as soon as EVC overflow happened at one process.

By looking at this chart, it is possible to notice how the number of events executed at a process increases in a nearly linear way in respect to the number of processes taking part in the execution. This can be explained if we look at the EVC overflow value (32*n), which is a function of the total number of processes. So, the higher is *n,* the greater will be the maximum allowed value for the EVC before stopping execution: this means that a greater number of events may be executed. This effect is anyways countered by a bigger prime number assigned to some processes, which will worsen the effect of local ticks on the EVC size. Evidently, this chart tells us that the overflow value increases more rapidly than how the local tick worsens the growth of the EVC size due to bigger prime numbers.

The number of events executed at a process increases by one every increase of 20-30 in the total number of processes taking part in the execution. Obviously, some outliers can still be noticed, but they are always in the range of ± 1 in respect to the average. Considering that the simulation was run for every single number of processes in the range [10, 200], we can anyways tell that the values are pretty stable most of the time, following a precise growth line.

During the third experiment, the objective was to try different percentages of internal events while also modifying the number of processes used in the simulation. The percentage of internal events was moved in the range [0,99] (where 0 means that all events will be send events, and 99 that pretty much all events will be internal), while the number of processes that were tried were 10, 20, 40, 60, 80, 100:



The data was gathered using the average set of values after 10 runs of the simulation for each one of the selected *n* values in order to minimize the effect of outliers. The execution of the simulation is stopped as soon as the EVC overflow value is reached at a process.

For each value of *n* we can observe a pretty similar behavior: the total number of events in the system grows slowly with the increase of internal events, and its growth speeds up very rapidly at around 80%-90% of internal events. All of this can be explained by thinking how the EVC can grow: with a low percentage of internal events, the majority of all events will be send events. This implies that a lot of receive events will be executed, and as such, many LCM will be calculated during merge operations. LCM is a great cause of EVC growth in size, and for this reason, not so many events will be executed, due to a fast reach of EVC overflow.

If, instead, we move towards higher percentages of internal events, a lower number of LCM operations will be executed. Therefore, the EVC will grow slower in size. The greater cause of EVC increase now will simply be the local tick. This implies that an increasing number of events can be executed in the system, having a slow EVC growth. This can be noticed particularly well in right part of the chart, where the number of events executed explodes, having pretty much only local ticks as the principal cause of EVC growth. All of this tells us that the receive operations, together with their merge operations, are the main cause of fast EVC growth.

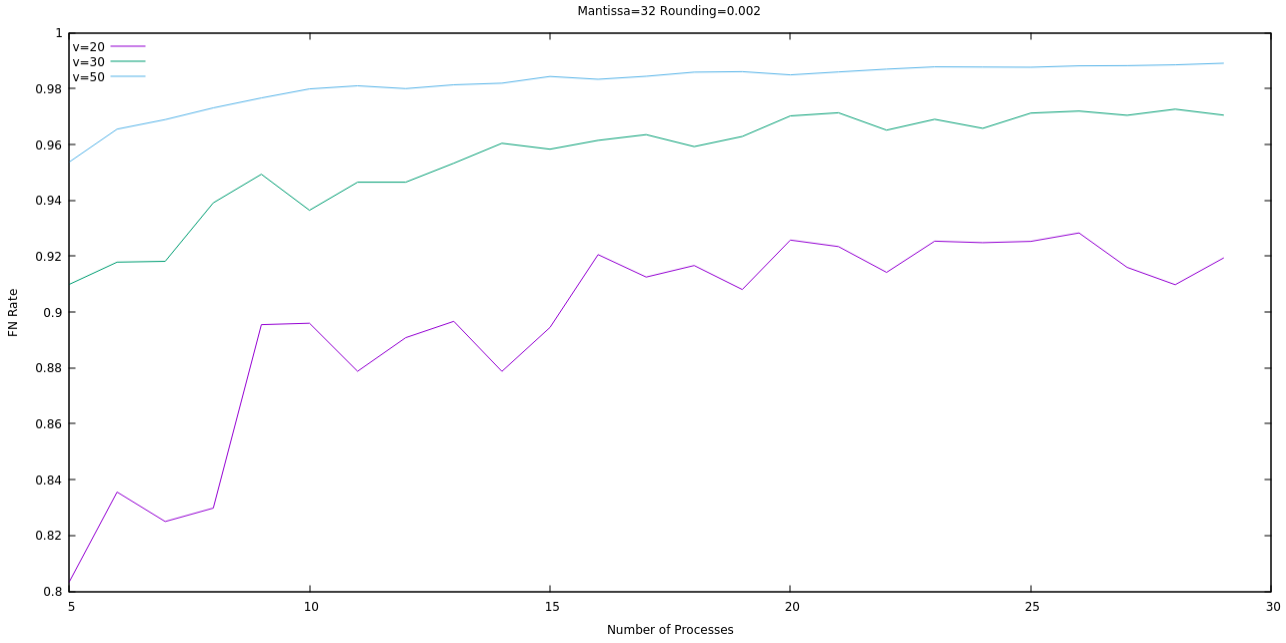
If we exclude very high percentages of internal events, we can observe that every process executes around 27-30 events before overflow. This result confirms what was already analyzed in the previous chart.

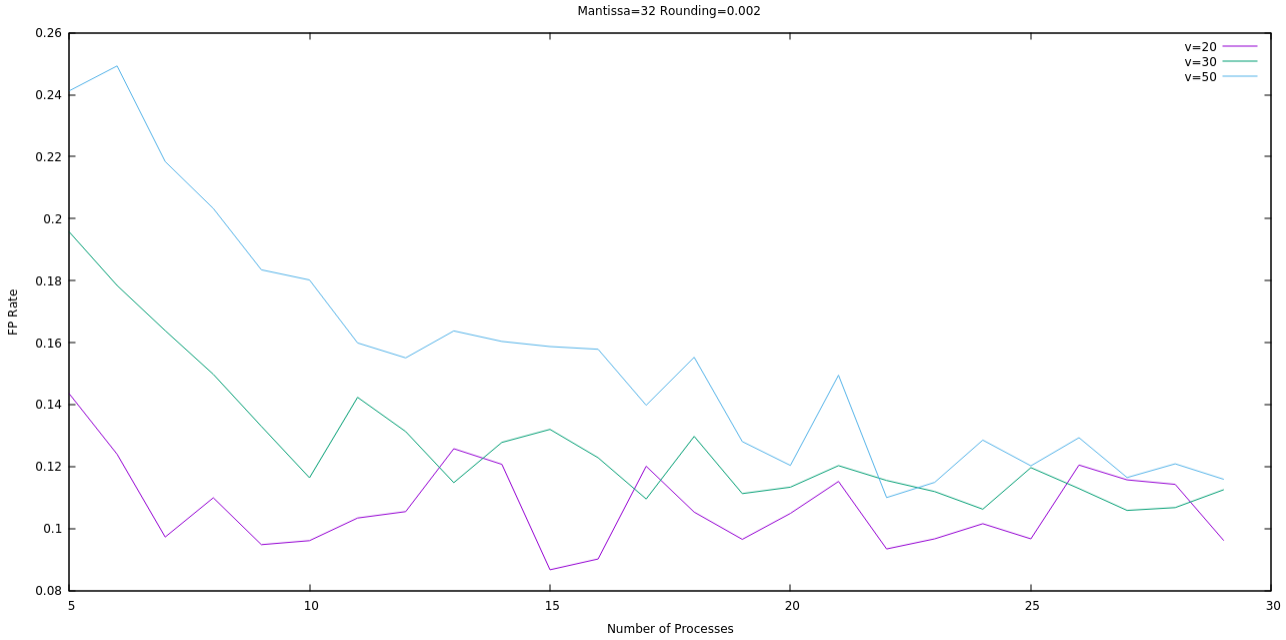
- **Part 2:**

In this second part of the analysis, the objective was to analyze the behavior of the logarithmic form of the EVCs. After having executed the normal simulation, each couple of executed events is analyzed in order to understand its causal dependency relation. Few factors are able to substantially modify the performance:

- N parameter: the number of processes directly influences the total number of events executed in the system and, as such, the EVC growth.
- V parameter: this corresponds to the number of events executed by a single process.
- M parameter: this indicates the mantissa size. It was considered as the number of digits that are present after the dot in a decimal number.
- Rounding parameter: this is the minimum sensibility for which a decimal number is approximated to an integer number. This means that, in order to check if a given number can be approximated to an integer, we first find the closest integer (lower integer will be chosen if decimal part is < .5, while greater integer will be chosen if decimal part is >= .5 e.g. closest integer to 5.3 is 5, while closest integer to 5.6 is 6), then we calculate the absolute value of the difference between our number and the selected closest integer. If the resulting difference is lower than the currently set "*Rounding parameter*", then the number is approximated with the selected integer.

During the first experiment, the number of processes was varied in the range [5, 30] while analyzing the FN and FP Rate values. All of the following charts were obtained with 50% chance of internal events (in respect to send events). The execution of the simulations is stopped once all of the processes have executed "*v*" events. In order to have a more detailed analysis, the simulation was executed again while varying other parameters, such as the value of *v,* the rounding factor, and the mantissa size.

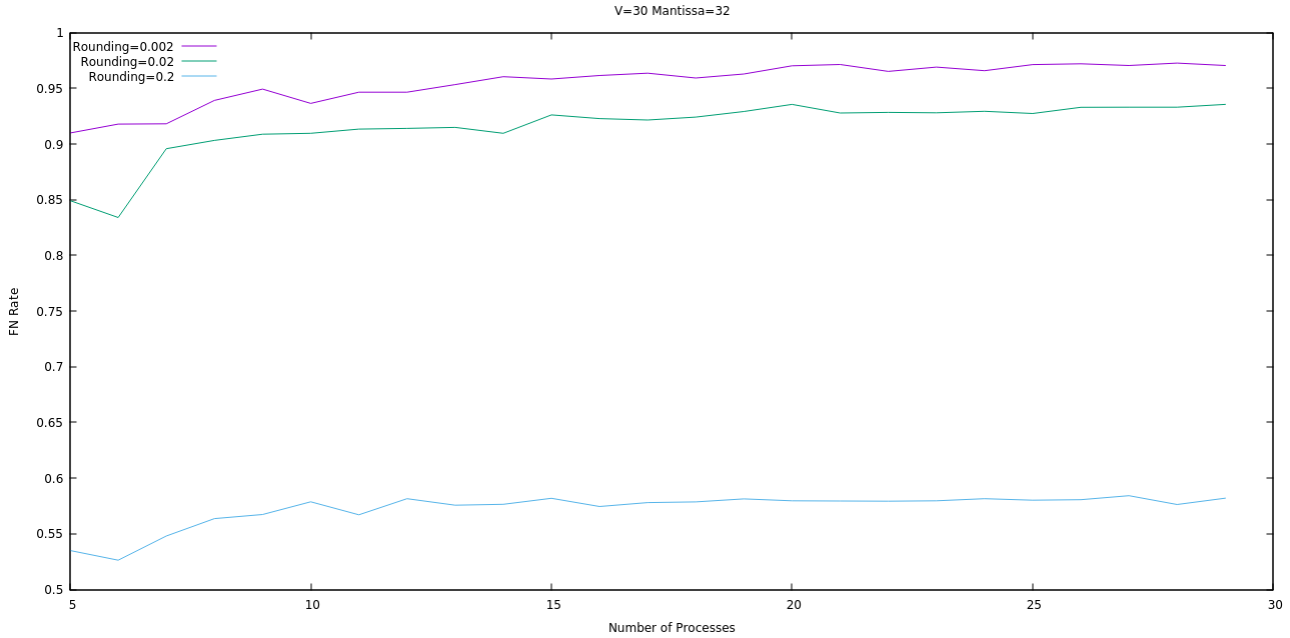


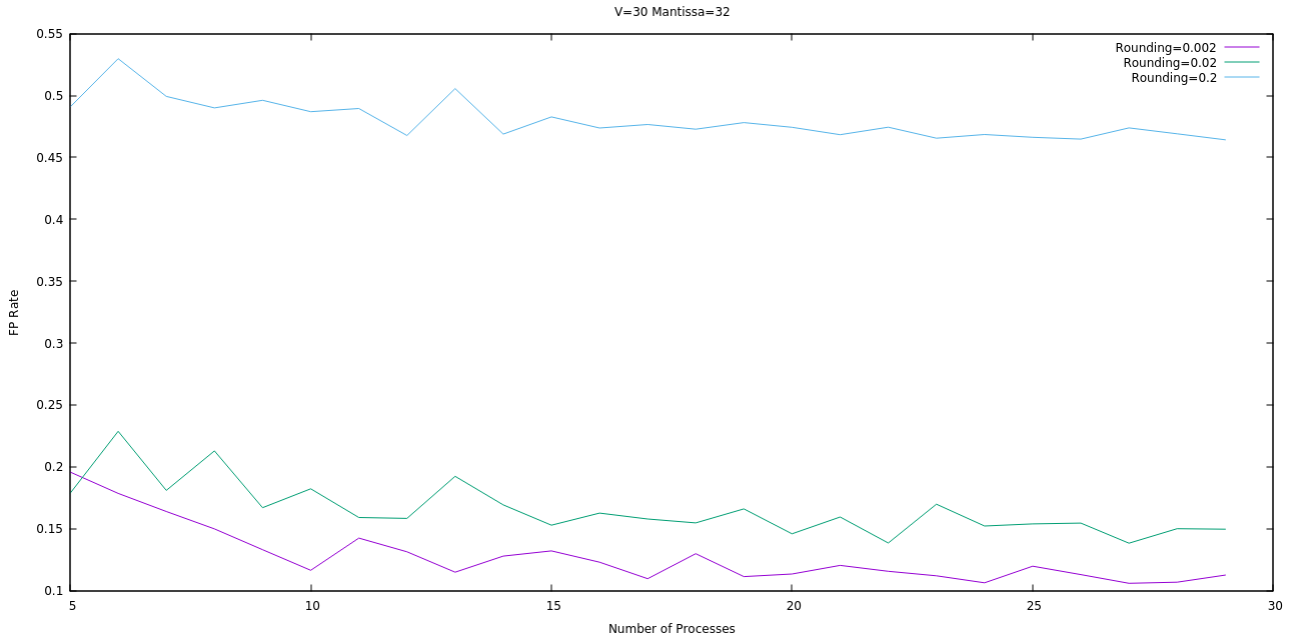


This first simulation was executed multiple times, varying the *v* parameter (v=20, 30, 50) in order to have a different number of events executed at each process. Mantissa was kept at a value of 32, while the rounding factor was 0.002.

The second chart, by showing a decreasing trend in the FP rate, hints to an increasing value of the True Negatives with an increasing number of processes. We can also notice how, in case of higher values of "v" (meaning a greater number of events more in general) the FP rate is higher for a low number of processes, but gets stabilized around 0.1 for every value of "v" that was chosen. Because of that, we can observe that for a reasonable number of processes, independently of the number of events executed at a process, we get a FP rate value that is stable around 0.1.

The first chart, instead, shows a pretty high value for the FN rate value. This trend seems to be very dependent on the value of "*v*", as can be seen that for v=20 the FN rate gets stabilized on a value that is nearly 0.1 lower than what happens for v=50. A possible interpretation of this could be that the number of false negatives increases with the total number of events executed in the system more rapidly than how the true positives do.



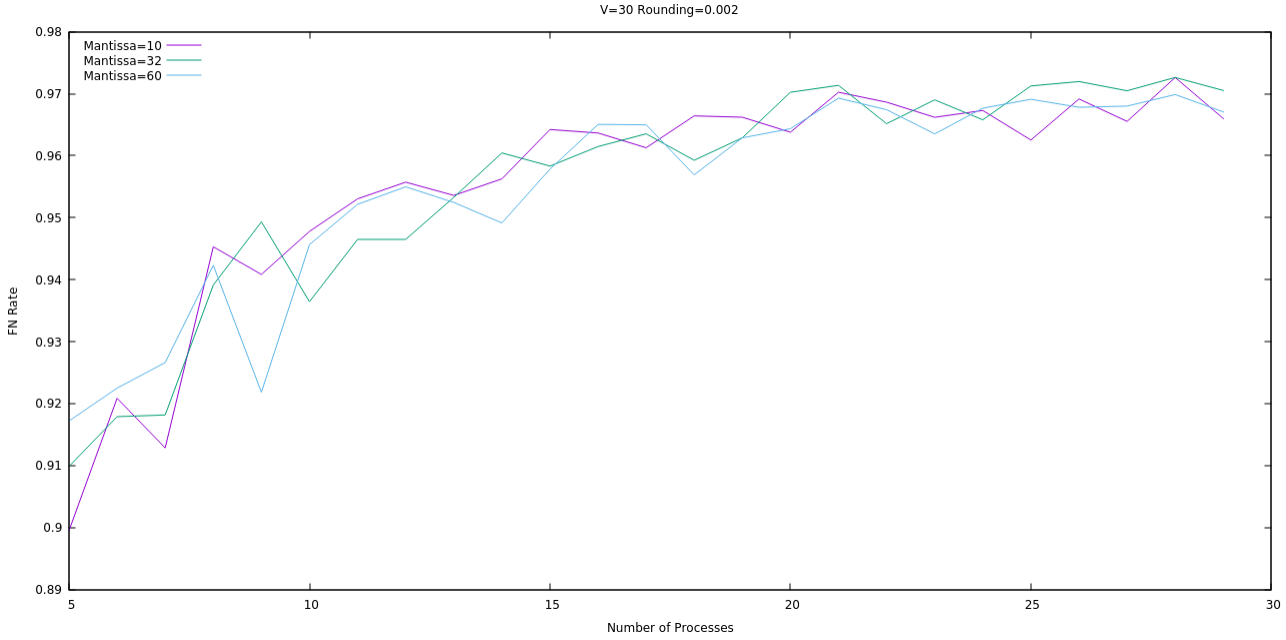


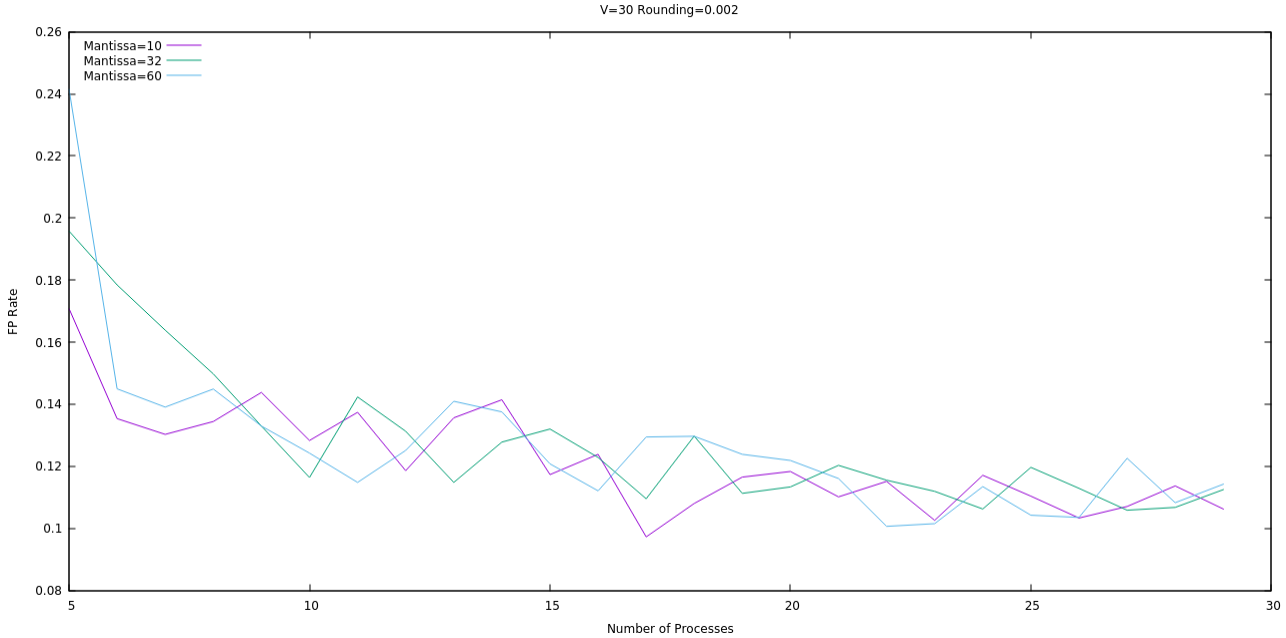
The same kind of simulation was then run while varying the Rounding factor (0.002, 0.02, 0.2). The *v* value was kept at 30 events per process, while the Mantissa was 32.

The first chart, similarly as before, shows an high value of the FN rate. Anyways, here we can notice how the rounding factor is able to influence a lot the performance of the system. For a more precise value of the rounding factor (0.002), only numbers that are actually very close to an integer will be considered integers. Because of that, a lot of inverse logs will not be considered integers, and as such they wont be part of the true positives. If instead we allow more numbers to be considered as integers by increasing the rounding value up to 0.2, more inverse logarithms will happen to be considered part of N, and as such they help the True Positives increase. The FN rate even comes down to a value of 0.55.

Having an high rounding factor such as 0.2 can help decrease the FN rate on one side, but has some side effects on the FP rate, as many more false positives will appear (many numbers will incorrectly

be considered as integers). This obviously increases the FN rate up to a value of 0.5. We can notice how the lower is the rounding factor (0.002), the lower is also the FN rate. All of this hints at the fact that a low rounding factor has to be chosen in order to favor precision at the cost of having high FN rates.



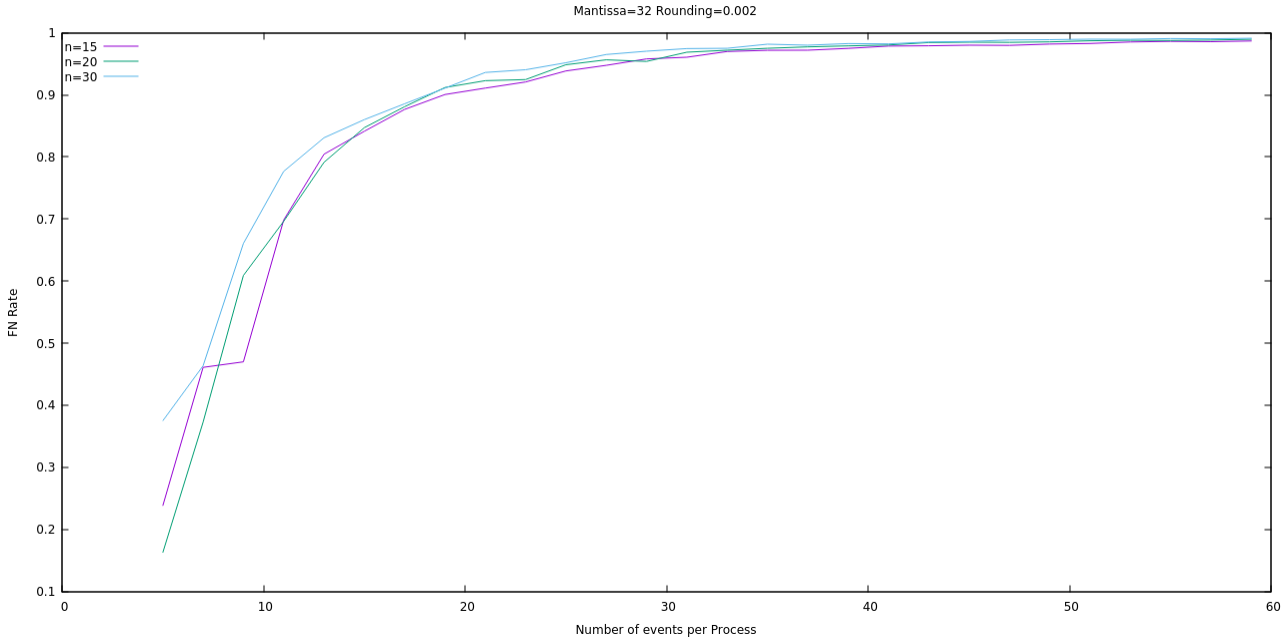


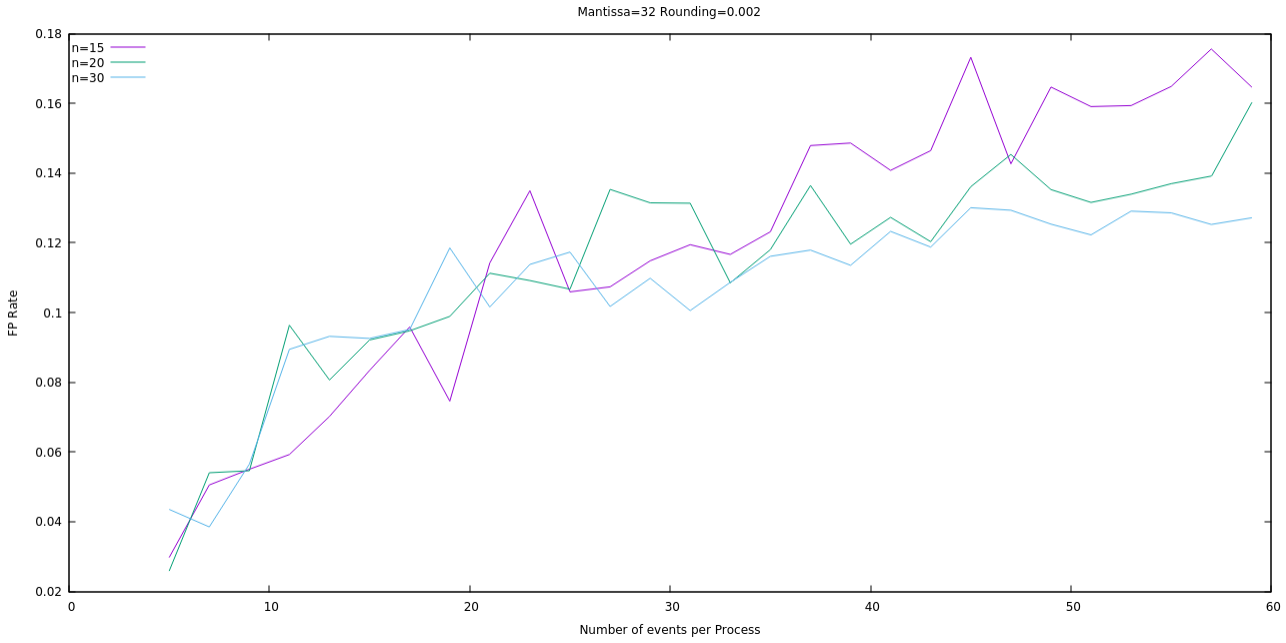
The last simulation for this first experiment was done while varying the mantissa size (10, 32, 60). The selected value for "*v*" was 30 again, and the rounding factor was 0.002.

As already pointed out for the other charts, the FN rate stays pretty high, while the FP rate is low. We can see that the effect of the mantissa size is not very noticeable: only that for a low mantissa value of 10 we have a bit higher FN rate values. This can be probably explained by looking at the rounding factor: it is the main performance modifier, as it is the one that determines when a number can be considered an integer: so, the greater precision given by a bigger mantissa is hidden under a

rounding factor that tries to reduce the precision itself in order to approximate numbers to their respective closest integers.

In the second type of experiment, the FN and FP Rates are calculated at the varying of the *"v"* parameter in the range [5, 60], which indicates the number of events executed at each process. All of the following charts were obtained with 50% chance of internal events (in respect to send events). The execution of the simulations is stopped once all of the processes have executed "*v"* events. In order to have more expressive results, also other parameters such as the number of processes *n,* the mantissa size, and the Rounding factor are varied for some executions (when explicitly indicated in the chart).



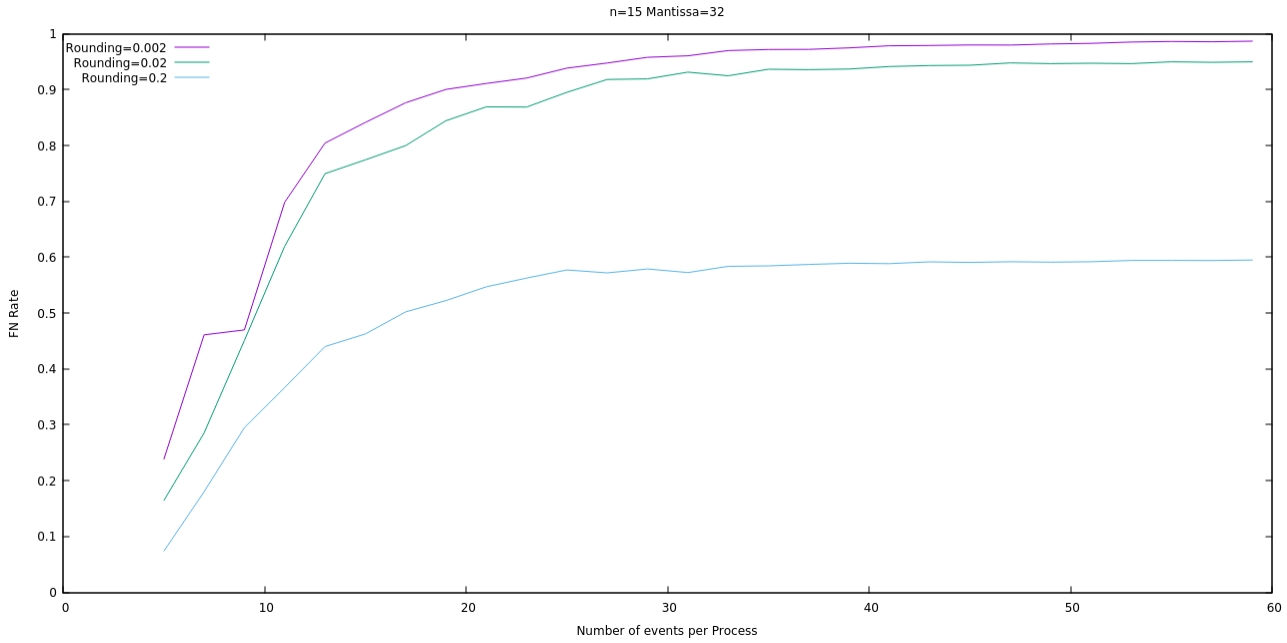


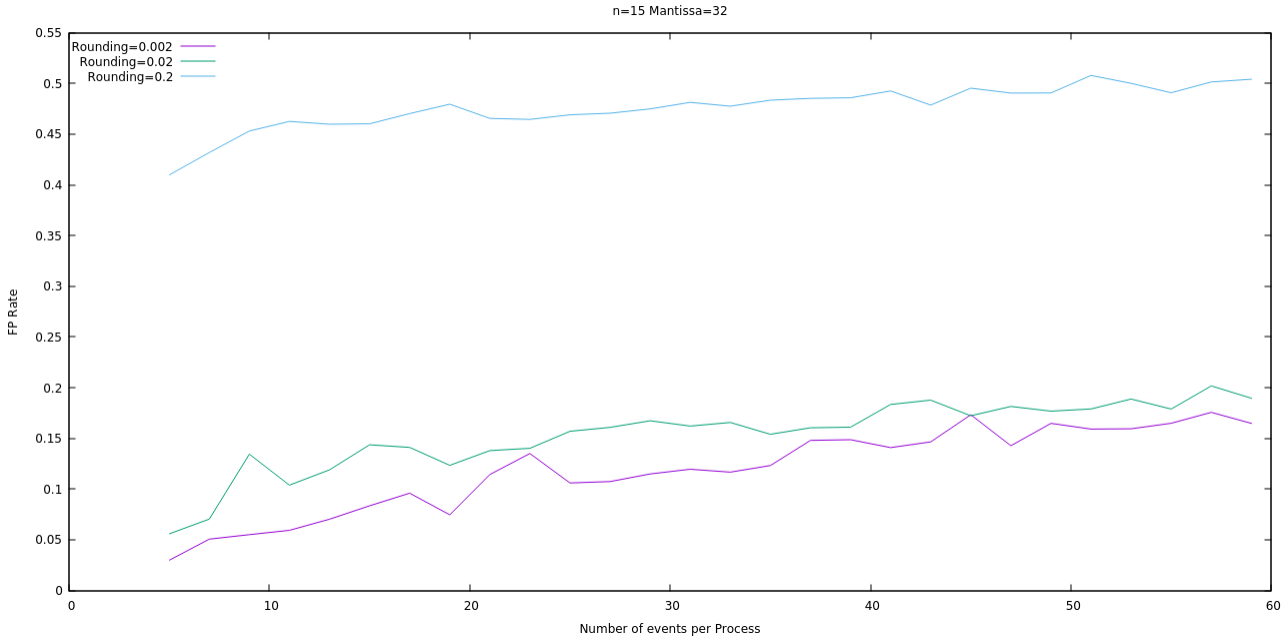
In this simulation, other than the *"v"* parameter, also the number of processes "*n*" in the system is varied (n=15, 20, 30). The mantissa is set to 32, while the Rounding Factor is 0.002.

The first chart shows that the FN rate increases rapidly for low number of events, and finally gets stabilized at very high values of FN (over 0.9), just as we've already observed in previous experiments. It is pretty evident that the number of processes taking part in the simulation does not influence particularly those values, as all 3 curves follow the same trend in a very precise way. This indicates that it is the number of events executed at process alone that is able to increase the number of False Negatives a lot.

When we look at the FP Rate chart, we can see a slower growth of the chart, followed by a stabilization just above an FP rate of 0.1. After this value, the curve does not seem to grow particularly anymore. The same behavior is respected by all three curves referred to the three values of "*n*". This probably implies that after a value of around 40 events per process, the number of false positives and true negatives grows at the same speed.

Thanks to those two charts we've noticed that it is not the number of processes that modifies the FP and FP rates as much as the number of events executed as a single process does.



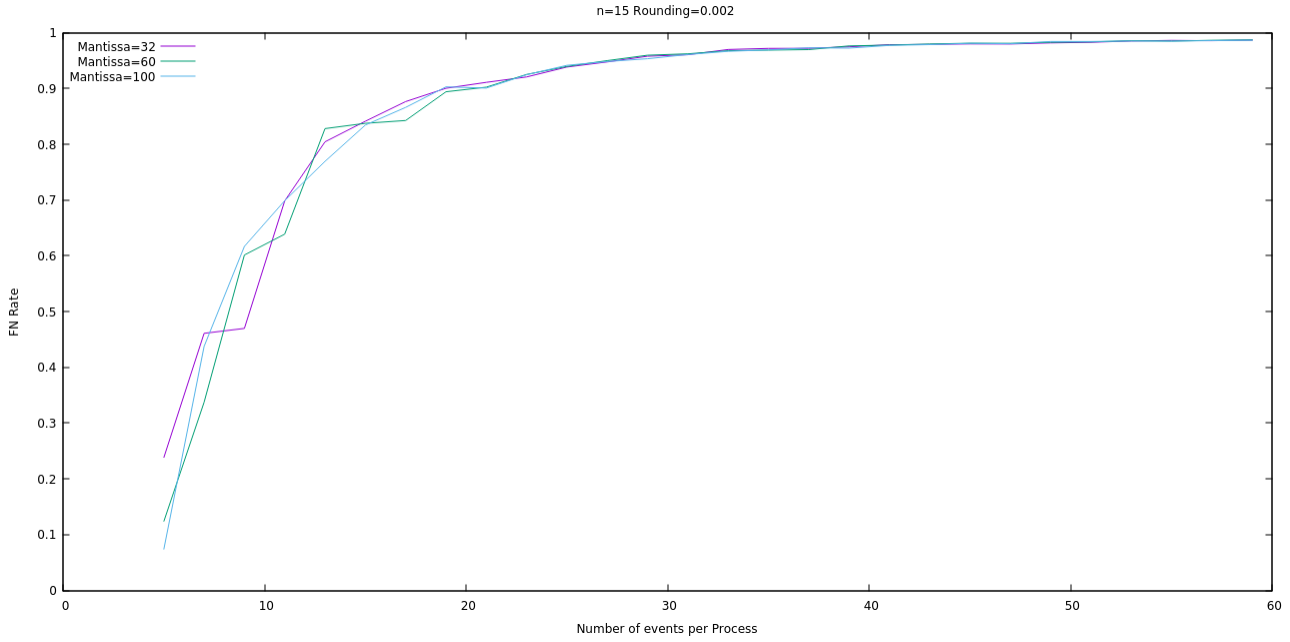


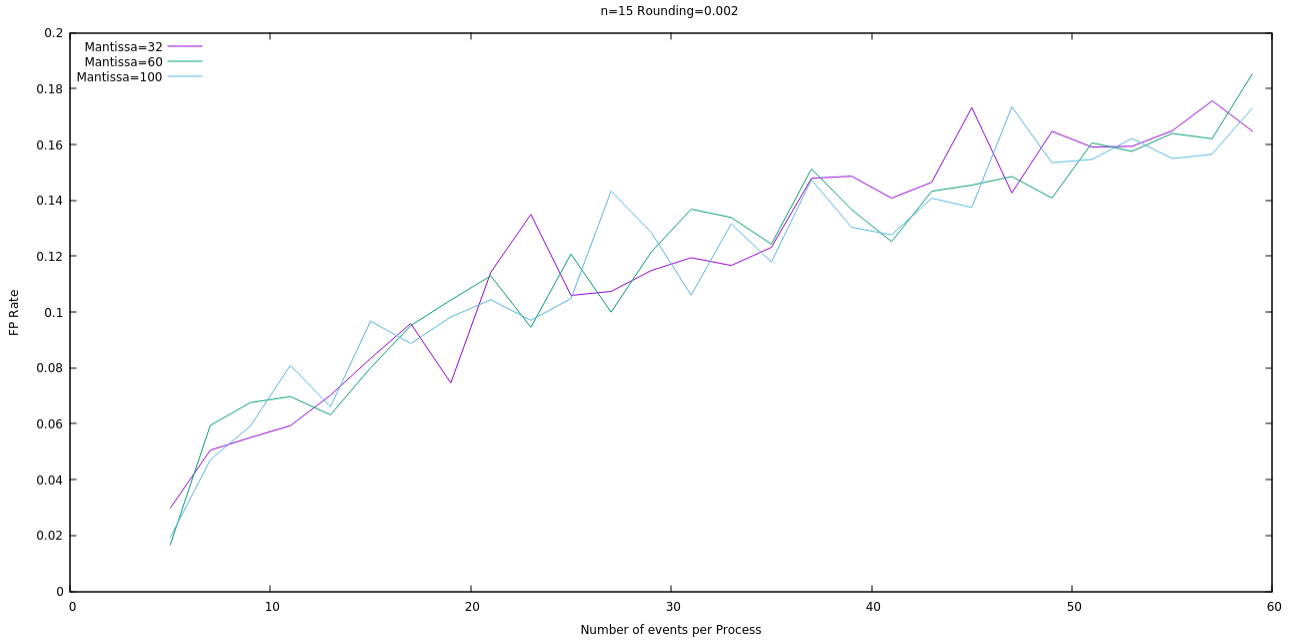
The second simulation was executed changing the Rounding Factor between 0.002, 0.02 and 0.2. The selected number of processes was 15, and the mantissa size was 32.

Looking at the first chart, in the opposite way in respect to the previous experiment, we can notice that the Rounding Factor is able to modify a lot the performance of the system. As we increase the rounding factor towards 0.2, the FN rate keeps dropping until 0.5. Instead, if we keep it smaller, the FN rate stabilizes at around 0.9 just like in the previous experiments. This behavior was already notice while changing the rounding factor in respect to the number of processes taking part in the simulation. The possible explanation is that a great rounding factor such as 0.2 allows a bigger number of inverse logarithm values to be approximated to integers. As such, the true positives will surely increase, and the FN rate will decrease.

On the complete opposite hand, the FP rates stabilize around the usual value of 0.1 – 0.15 for small values of the rounding factor. But as we increase it towards 0.2, the FP rate increases as well, up to a value of 0.45. This is also a behavior that was noticed previously. For the same reason as before, having a lot of inverse logarithms to be considered as integer numbers, allows for a bigger chance to catch true positives, but at the cost of having many false positives too.

This indicates again that the Rounding factor has a big role in how the FP and FN rates move around. A good rounding factor is one which is not too big to consider as belonging to N too many inverse logarithms even in wrong cases, but that is also not too small to the point of not considering as integers enough numbers (as this might kill the true positives).



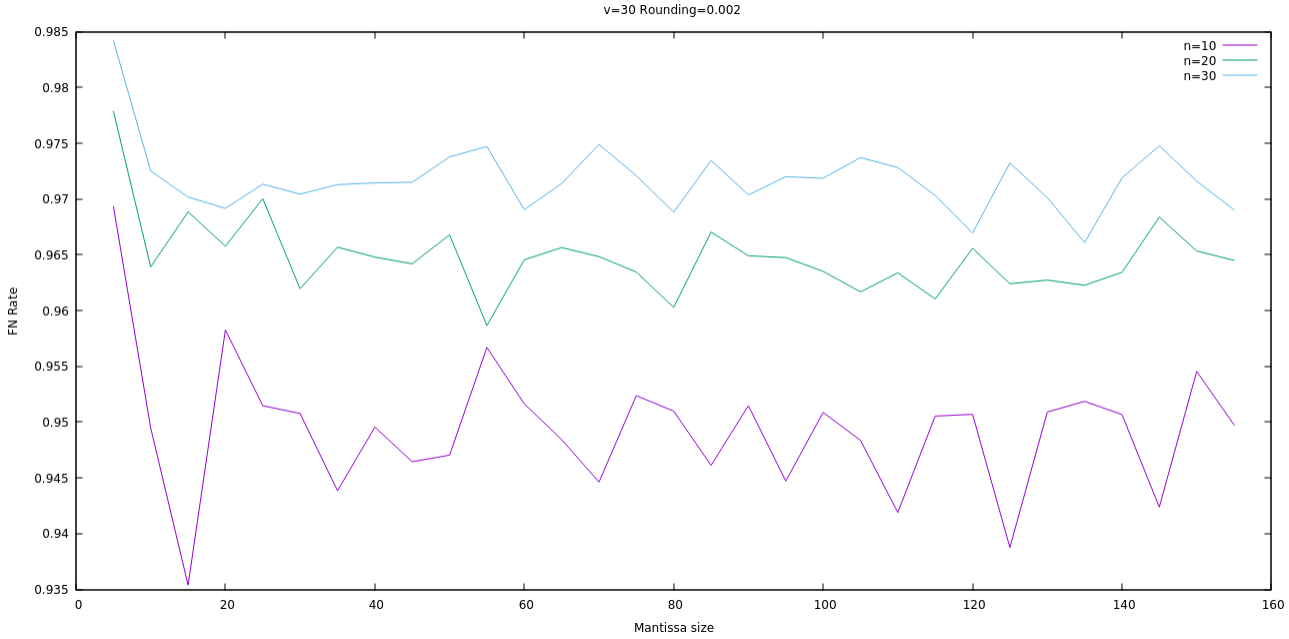


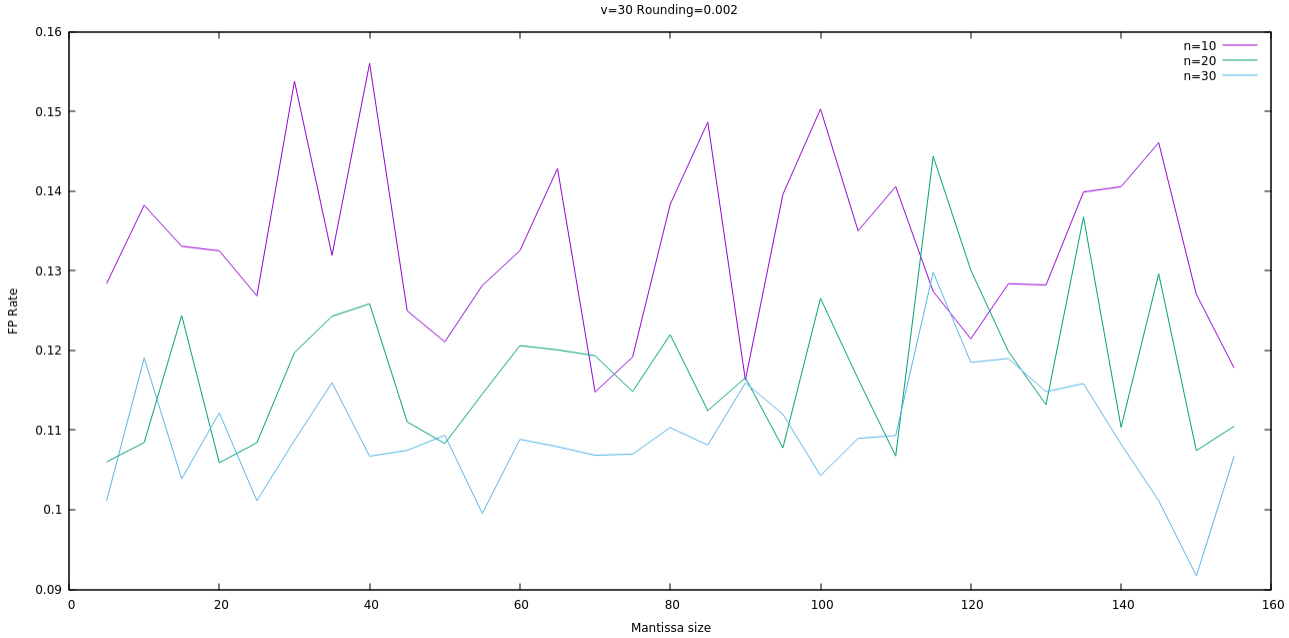
In this last simulation, the mantissa value was varied in the values of 32, 60, and 100. The number of processes was set to 15, and the Rounding factor was 0.002.

In both charts we can immediately see that the mantissa size does not modify substantially the behavior of the curves. In the first chart, the FN rate gets stabilized at around 0.9 after about 30 events per process. In the second chart instead, the values get stabilized at around 0.15.

This is because a lot of the precision that is piggybacked by the increased mantissa size is lost during the rounding process. In order to preserve this precision, we should lower the Rounding factor even more, at the cost of increasing even more the False Negatives. The benefits of having the mantissa carrying more precision might not be worth against a bigger rounding factor that is able to reduce more substantially the FN rate, at the cost of a small FP increase.

In the third type of experiment, the FN and FP Rates are calculated at the varying of the Mantissa size in the range [5, 160]. All of the following charts were obtained with 50% chance of internal events (in respect to send events). The execution of the simulations is stopped once all of the processes have executed "*v*" events. In order to have more expressive results, also other parameters such as the number of processes *n,* the "*v*" parameter, and the Rounding factor are varied for some executions (when explicitly indicated in the chart).





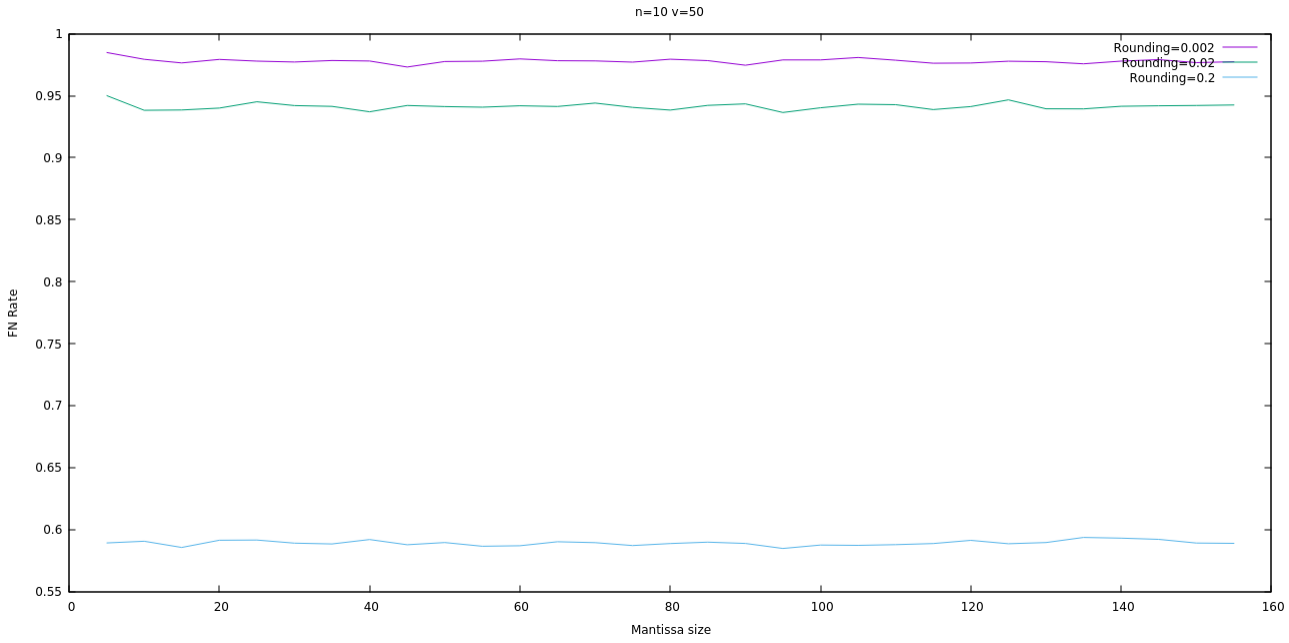
In this first simulation, the number of events *n* has been set to 10, 20, and 30. The "*v*" parameter was set to 30, and the Rounding factor was 0.002.

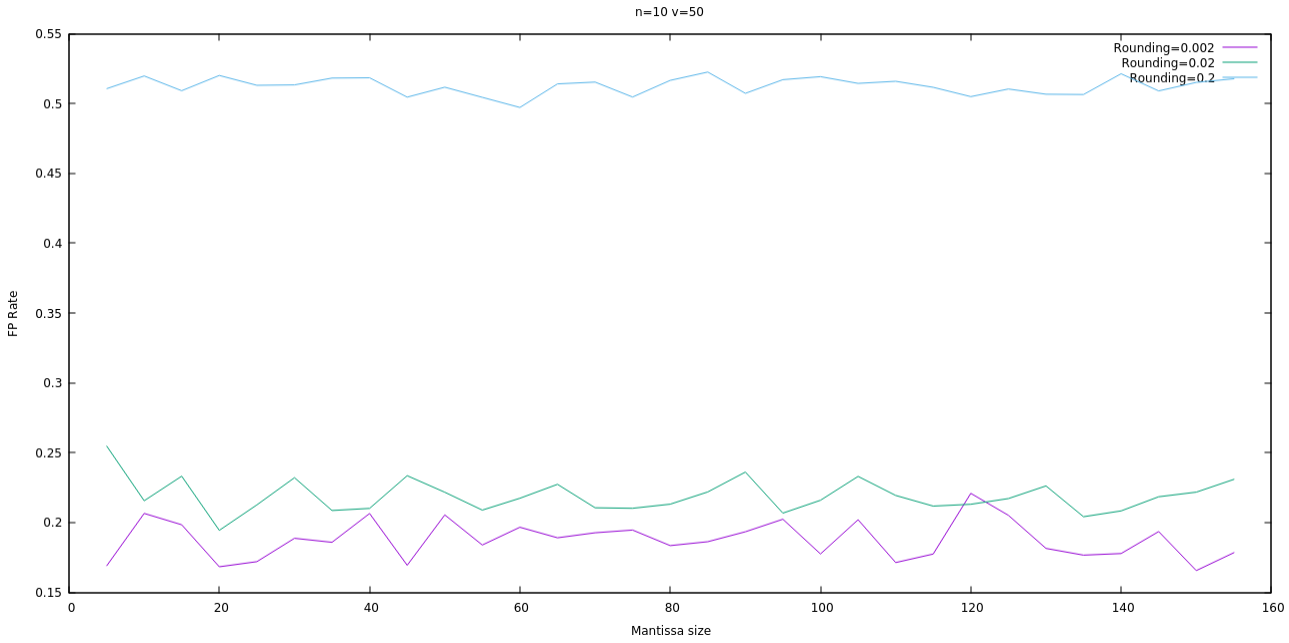
In the first chart, we can see again that for high values of *n,* the chart stabilizes at high FN rate values. If the number of processes is reduced, then the FN rate value drops only by about 0.05. That is because, as we've already seen in the second experiment, the number of events *v* is the main player

in determining those rates with a greater influence. More in general, we can notice that for low mantissa values (under 20), there is a big drop of FN rate, then followed by a constant trend.

The FP Rate chart shows a similar behavior, but we don't have the same initial instability for low values of mantissa. The FP rate tends to remain constant at about 0.12.

Following the same observation as before, the FP rate is just a bit higher for lower values of *n,* but just for a value of 0.02 in respect to higher ones.





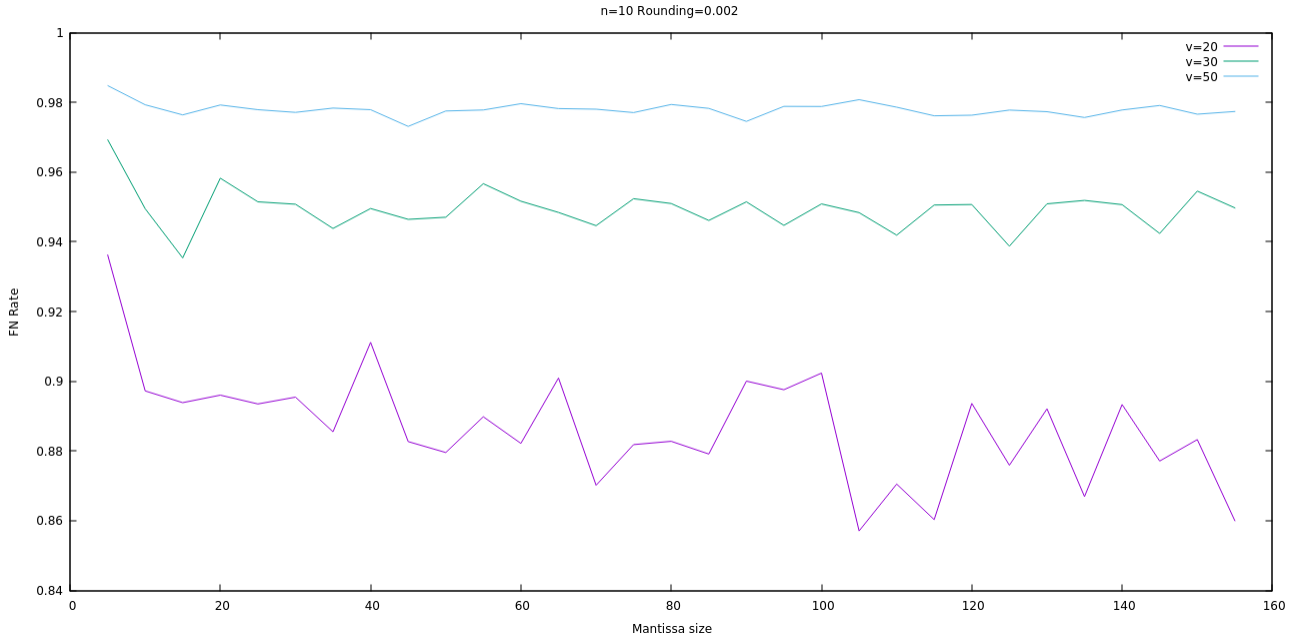
In the second simulation, the rounding factor was changed between 0.002, 0.02, and 0.2. The number of processes was set to 10, while the *v* parameter was set to 50 events per process.

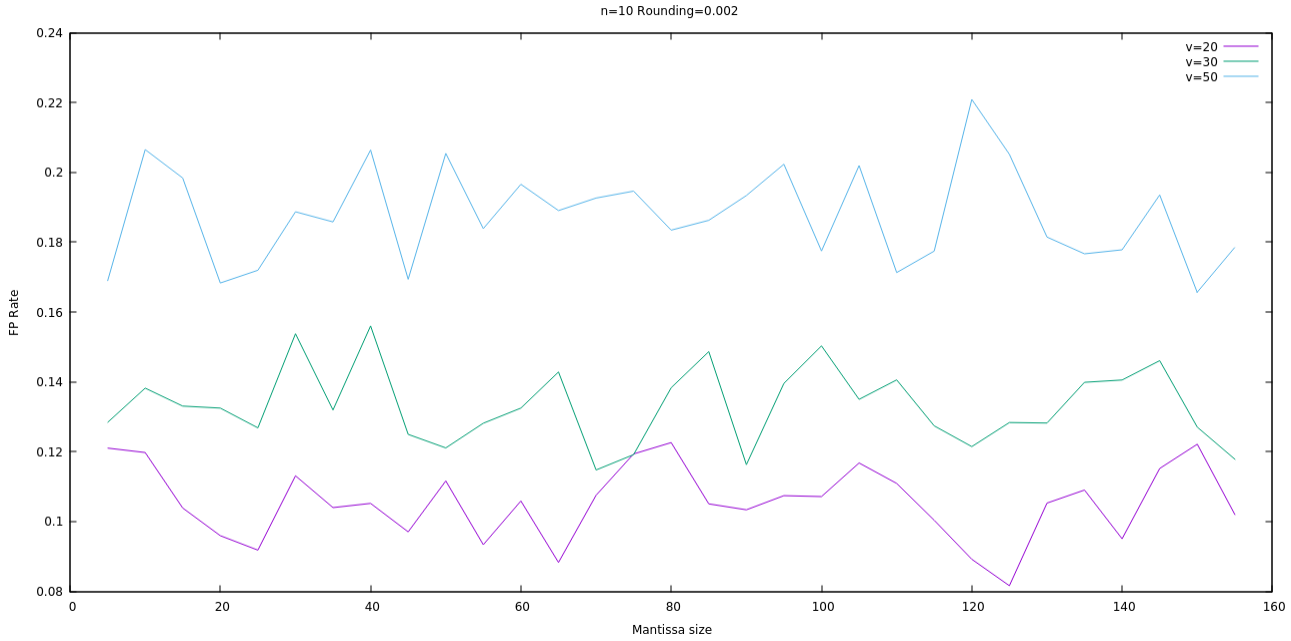
One of the first things that we can notice here is how constant the FP and FN rates remain with the varying of the mantissa size. As already underlined by some of the previous experiments, this might

be because the precision carried by a greater mantissa size is often reduced by the rounding factor necessary to improve the number of true positives (at the cost of false positives).

In fact, the first chart shows that a greater rounding factor such as 0.2 is able to reduce the FN rate down to 0.6, but actually increases the FP rate up to 0.5. This is because way more inverse logarithms are considered as belonging to N.

All of this suggests that the Mantissa size starts to matter when an extremely low rounding value might be chosen.





During this last simulation, the value of the "*v*" parameter was changed between 20, 30, and 50 events per process. The number of processes was set to 10, and the rounding factor was 0.002.

At first glance, we can notice how again both the FN the FP rates tend to remain constant with the mantissa size. The only exception can be found in the first chart, where for a small value of the "*v*" parameter (v=20), we notice a decreasing trend of the false negatives.

Other than this, we can notice again how for greater values of "*v*", we have greater number of false negatives, as well as higher number of false positives. This might be due to the fact that probably false negatives and false positives are able to grow faster than how true positives and true negatives do. This behavior is evidently accentuated by an higher number of events.