Course CSE 532: Multiprocessor Architecture
Authors: Rishabh Jain, Niramay Vaidya
Project Title: Performance Analysis of Association Rule Mining

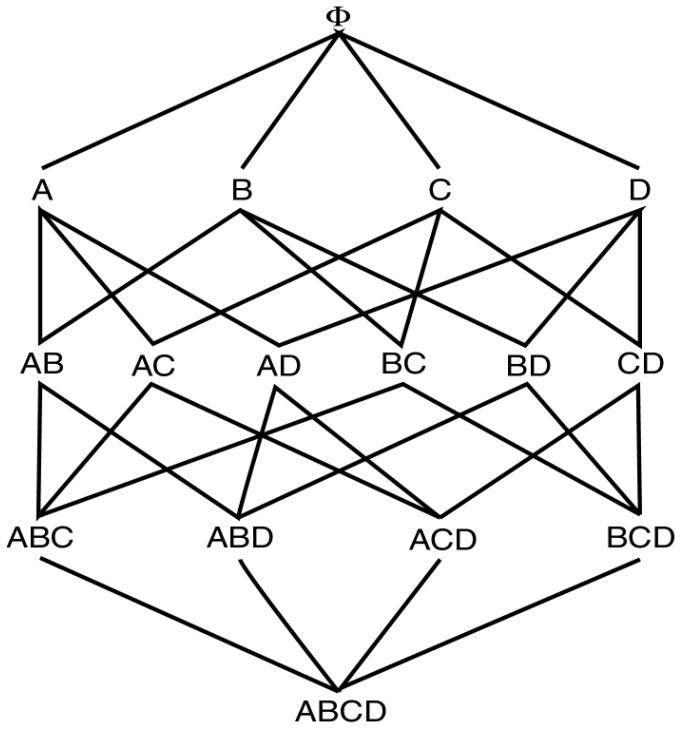
Note: Both authors have contributed equally (equal division of work between the three implementations and corresponding analysis)

GitHub repository: https://github.com/rishucoding/532-perf_analysis
Note: CUDA code is available in branch simple_bitmap whereas the pthreads and serial code can be found in the main branch

Outline:

1. Quick pointers on the application

[2] mentions that determining the buying patterns of demographic groups or segmenting customers according to relationships in their buying patterns is called data mining. A particular type of data mining is mining for associations. The goal here is to discover relationships / associations using the available information between different customers and their transactions and to generate rules for the inference of customer behaviour. In more concrete terms, this problem entails sets of ‘k’ items that are found to occur together in more than a certain threshold fraction of the transactions, given a database / dataset. Such a set of items would then be identified as a frequent itemset. In order to do this, a property that an itemset is frequent if the itemsets it was formed from (those in the previous level) are frequent can be utilized. Using this, the problem essentially boils down to performing computations in the manner shown below (taken from [1]), in order to exploit this property-



Grouping items using incremental lexicographic ordering at each level is called equivalence class partitioning. Such a partitioning helps ease the computation for determining the candidate itemsets at the next level.

Note: The only difference in our implementation is that instead of using alphabets to represent items, increasing positive integers have been used.

2. Dataset

2.1. Structure

A typedef struct is used to define an entry in the transaction database. This entry includes two fields: a txid to store the id of the entry, and an int array to mark all the items present. The upper limit on the total items is presently 100 and it is parameterized. The total number of entries is defined by a constant NUM_TX, and the total number of items is defined by a constant NUM_ITEMS. Note that the database can be visualized as a 2D array (matrix) where each row represents a transaction timestamp, and each column indicates the items purchased in that transaction.

2.2. Dataset generation

A method called **"generate_dataset"** is used which populates each cell in the matrix. A uniform probabilistic distribution is used to decide the chance of purchasing an item in any transaction entry. The value of chance is parameterized. Having a low chance gives a sparse matrix while a high value of chance gives a dense matrix.

For the special case datasets (having particular distributions representing skewness in data), a python script has been written to generate 4 different matrix variants of a particular size.

2.3. Parameters under control:

a) NUM_TX - this represents the total entries in the database
b) NUM_ITEMS - this represents the total columns/items in the database.
c) THRESHOLD - this represents the minimum support
d) Probability - For a given transaction entry, this determines the chance of purchasing an item.

2.4. Practical relatability (making sense in the real world)

1. During situations like off-season, holidays, high inflation, etc, the purchase of items could go down, and thus, each transaction would have a lower number of items marked. Thus, for this case, a sparse matrix is an appropriate representative.
2. On the other hand, when there is a high demand or right season for certain produce, like black friday deals, winter sales, etc, people often tend to buy in bulk, and stock up things. This would mean a high number of items marked per transaction. Thus, a dense matrix appropriately represents this case.
3. Threshold decides our association rule. For example, a 33% threshold means that we want to study all combinations of items which occur on average in every 3rd transaction. This parameter could help the sales expert in deciding what items are being frequently purchased, and thus give an opportunity for introducing varieties. Like, if milk is purchased frequently, a wise choice would be to introduce flavoured milk, almond milk, etc. However, if some item is not being purchased frequently, like steel utensils or study lamps, it could mean that these items serve for a long time.

3. Implementation specifics

3.1. Naive sequential (1 core)

As explained in [2], the naive way to solve this problem is to traverse the entire dataset to determine L1, by counting frequencies of all itemsets of size 1. From L1, the candidate itemset C2 can be computed. Using C2, the entire dataset can again be traversed to determine L2. This process can be continued as is further to generate all the frequent itemsets at each subsequent level. This algorithm has high computational complexity as it involves traversing the entire dataset for each candidate itemset to determine if it's frequent or not.

3.2. Parallelizing naive (multi-core)

According to [2], the way to parallelize this problem is to compute up to L2 serially, and then based on the equivalence classes present at the 2nd level, launch one thread for each equivalence class and beyond this point, each thread will keep on asynchronously generating frequent itemsets at each level. Every thread will perform the same mechanism henceforth as explained above in the naive sequential section.

3.3. Smartly doing sequential (1 core)

An algorithmic improvement over the naive sequential method is to compute as before up to L2, and then restructure the dataset such that it is now represented in the forms of lists of transactions for each itemset within L2. This saves traversing the entire original dataset for each subsequent level. This transformation of the dataset helps in combining the generation of the next level's candidate and further, frequent itemsets, from each previous level's frequent itemsets.

Steps:

1. A vector of vectors of int is used to represent each Li (frequent itemsets at each level). The inner vector contains items in each itemset and the outer vector represents the total frequent itemsets at each level. For each level, the length of the inner vector increases by 1.
2. A vector of sets of int is used to represent the restructured dataset for each level. The inner sets contain the txids for each corresponding frequent itemset at that level.
3. A vector of indexes is required to be computed at the start of each level to determine the equivalence class boundaries in order to determine candidate itemsets from each equivalence class for the generation of the next level of frequent itemsets using this equivalence class.
4. To determine if a particular itemset at the next level is frequent, a set intersection operation is done on the txid sets of the corresponding current level's frequent itemsets using which this next level's itemset has been formed.
5. To form this next level's itemset in question, a set union operation needs to be performed.

6. Finally, the **"compute_li"** function iteratively performs the generation of the next level frequent itemsets and outputs the above mentioned data structures for the next level, by taking as input the previous level's similar data structures.
7. C++'s STL (Standard Template Library) has been used extensively to perform all the operations mentioned above. The chrono utility has been used to measure the execution time.

Phases of development:

1. Initially, We figured out the computation of L2 and reformation of the dataset. **(commit ID- `7db323f`)**
2. Following this, we determined what data structures would be needed which could generically be used for all levels and implemented the L3 computation. **(commit ID- `f4d4a9a`)**
3. Then we extended the logic for L3 to compute all frequent itemsets at all levels. **(commit ID- `f058864`)**
4. Finally, we added the measurement of execution time. **(commit ID- `28f0350`)**
5. Once the smart algorithm was in place, we added support for the naive algorithm. **(commit ID- `400280c`)**

3.4. Parallelizing smart (multi core)

The only difference with respect to the naive thread implementation here is that as per the smart sequential mechanism explained above, each thread has all the data now local to it (i.e. the required txids for the frequent itemsets corresponding to the equivalence class it is responsible for).

Phases of development:

1. Figuring out how to parallelize from L2 based on each generated equivalence class by assigning a thread to each was straightforward. Certain modifications involving the need of a target function to be provided to the pthread_create call were required, but most of the other logic remains the same as that present in the sequential case. **(commit ID- `ae670fd`)**
2. We then added support for setting cpu / core affinity to each thread in a round robin fashion based on the number of available cores. **(commit ID- `9372d67`)**

3.5. Running on the GPU (accelerator)

The levels L1 and L2 are generated on the CPU. Further levels (L3, L4, …, etc) are generated on the GPU. As mentioned above, in the CPU, the data for the levels are stored as vector of vector, and corresponding transaction ids are stored as a vector of sets. Since the input and output data sent to the GPU is a simple 1D array, there is an additional data processing step before and after the kernel launch. This data processing involves two parts: (1) linearizing the present levels vector's to prepare the input data to the GPU kernel (2) converting the output array from the GPU into next levels vector's.

Steps:

1. Each level is computed in one iteration of the for loop.
2. For each level, an index array is calculated by the CPU to identify all the equivalence classes.
3. GPU kernels are launched for each equivalence class in the present level, where a GPU thread runs a cuda kernel to evaluate one candidate pair. This allows all candidates to be evaluated in parallel.
4. If the candidate pair is valid, it's corresponding next level data is stored in the output array. This computation involves intersection and union operations on two sub arrays.

Phases of development:

1. After figuring out that the cuda kernel can compute each level one by one, the first choice was to launch only one kernel per level which works on one equivalence class. This allows all the equivalence classes to be evaluated in parallel. However, when larger database configurations were set, this approach requires a lot of data transfer between the devices which lead to segmentation faults. **(commit ID- f1d691bc)**
2. Further improvement in (1) was to change the indexing strategy when writing to output arrays. This helped in covering some large database configurations, but it still suffered from segmentation faults. **(commit ID- 101d6db)**
3. This motivated us to think of a new way to parallelize the computation. So, instead of parallelizing for all equivalence classes, it was decided to parallelize for all the candidates within one equivalence class. This also gives more parallel opportunities as many times one equivalence class has 1000's of candidates to evaluate. **(commit ID- b36c171)**
4. Features of unified virtual memory and asynchronous prefetch were used. **(commit ID- 2b224c1)**
5. **nvprof** profiling tool was used to study different metrics like data transfer size, cuda API calls with individual time taken, and number of CPU page faults.

4. Correctness verification

As a first step, an initial correctness verification of all the 3 implementations (sequential, pthreads and cuda) was done on a very small database (10 transactions, 8 items, threshold = 3). This was achieved by comparing the generated outputs of each level with a manually calculated version of the outputs.

Once this was successful, to verify correctness of the implementation for any given dataset, a comparison among output files was made. This includes file size and the total count of the generated frequent itemset, from which it was concluded that all implementations are in sync, since these values turned out to be the exact same.

5. Measurement study

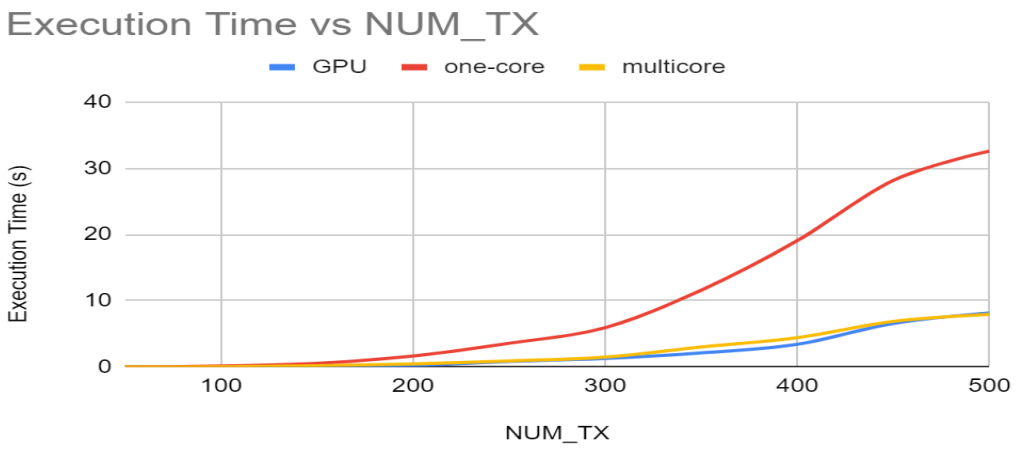
5.1. Timing measurement

1. All the obtained execution time results have been averaged over 3 measurements to ensure more accuracy.
2. A stopwatch style start and stop measurement is taken using the chrono c++ library by surrounding the start/stop around any computation block-of-interest.

5.2. Varying NUM_TX (number of transactions / txids)

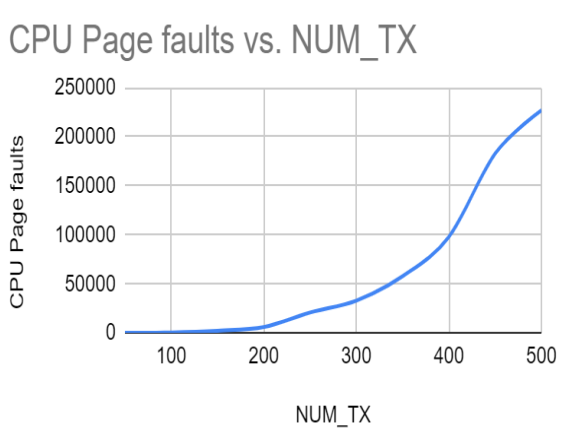
NUM_TX	NUM_FREQ_ITEMSETS	MAX_LEVEL_REACHED
50	62	L3
100	1680	L4
150	8240	L5
200	21317	L5
250	57913	L6
300	97825	L6
350	177942	L7
400	272667	L7
450	445848	L7
500	532578	L7

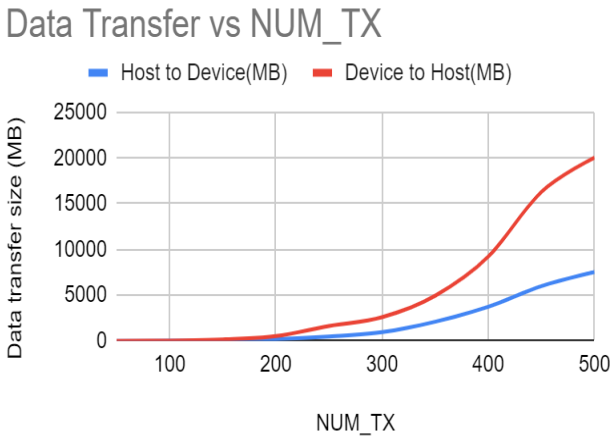
The above table shows the range of NUM_TX [50, 500] at a step size of 50 with the corresponding count of total frequent items sets and maximum level reached. The other parameters are fixed, particularly NUM_ITEMS = 50, probability = 50%, and threshold = 20. It can be seen that with an increase in the number of transactions, the total frequent sets and max level increases. This is because more transactions gives opportunity for more associations.



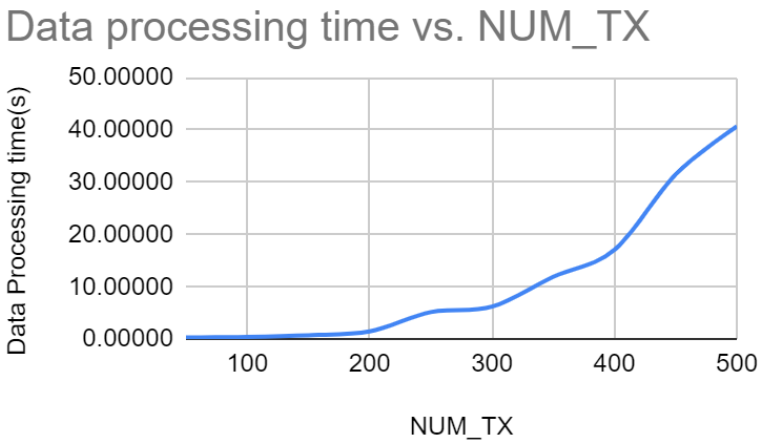
The above figure shows the variation in execution time as the number of transactions increases. It can be seen that the execution time increases with increase in NUM_TX. Both multi core and GPU perform much better than single core. At NUM_TX = 500, both multi core and GPU perform 4 times better than single core. There is not much performance difference between multi core and GPU. This could be because of the multi core implementation taking good advantage of 48 CPU cores in ladon server.

The next three figures discuss the profiling of CUDA implementation.





The above figure shows the variation in CPU Page faults and Data transfer as the number of transactions increases. Since unified memory is used, CPU Page faults occur whenever GPU accesses memory. The device to host memory size is higher than host to device because the output array size considers all candidates to be counted for the next level. Also, many times, the outputs are spatially far, thus falling into different pages, and leading to more data transfer. It is observed that the data transfer size is very high(like in GBs) when NUM_TX is large.



The above figure shows the variation in data processing time as the number of transactions increases. This processing is the same as mentioned in section 3.5 which happens before and after every kernel launch. In the present implementation, this is actually an overhead to use

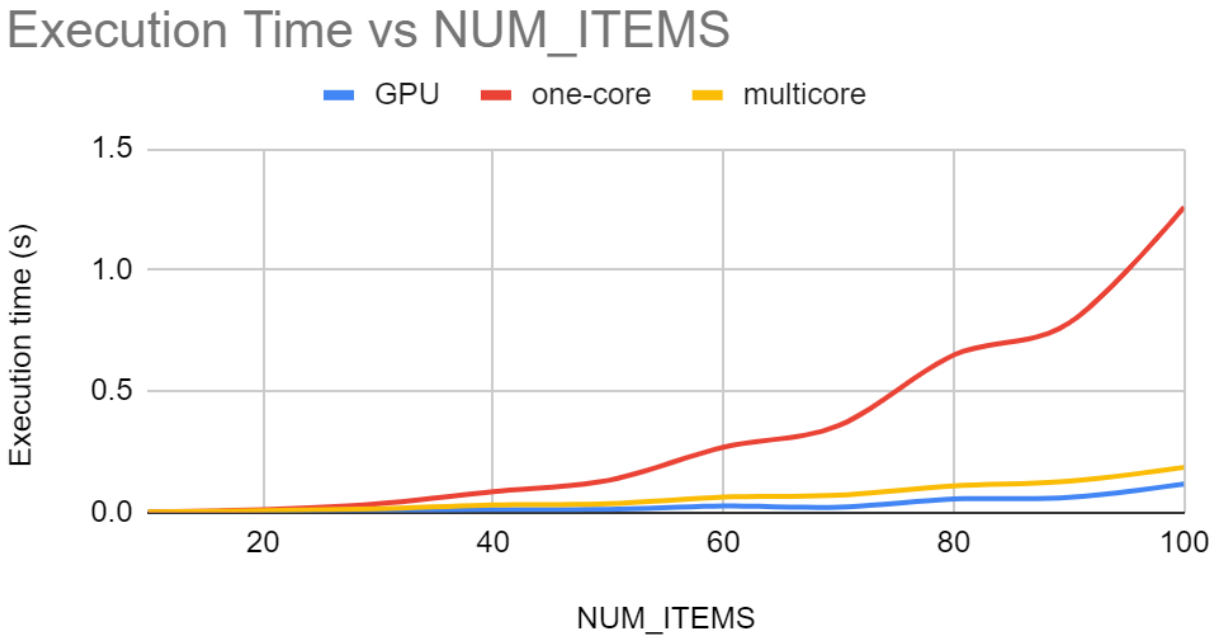
GPU. We believe this can be reduced significantly by cleverly designing the input and output parameters for the CUDA kernel.

5.3. Varying NUM_ITEMS (number of items)

Number of frequent itemsets and the highest level reached with varying number of items (along with the thread count in case of the pthreads implementation, based upon number of equivalence classes at level 2)-

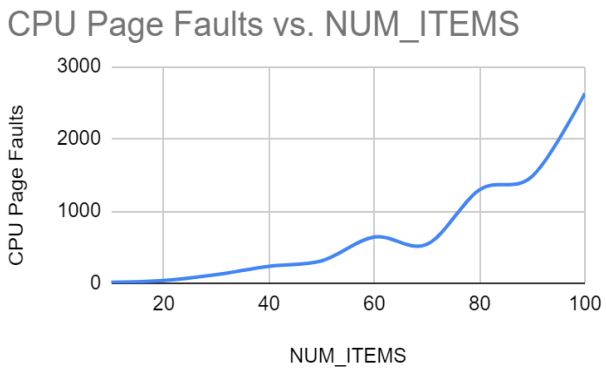
NUM_ITEMS	NUM_FREQ_ITEMSETS	MAX_LEVEL_REACHED	THREAD_COUNT
10	47	L3	9
20	211	L4	19
30	532	L4	29
40	1032	L4	38
50	1400	L4	49
60	2591	L5	59
70	2664	L4	69
80	5206	L5	79
90	5704	L4	89
100	9521	L5	99

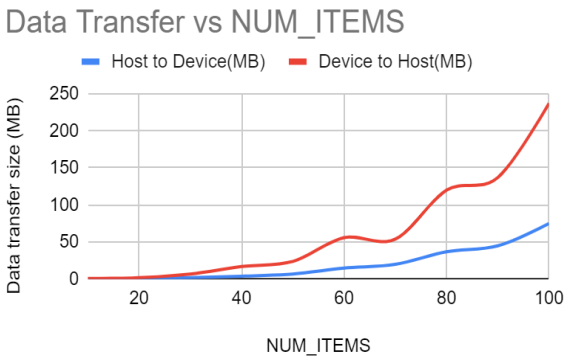
The above table shows the range of NUM_ITEMS [10, 100] at a step size of 10 with the corresponding count of total frequent items sets, maximum level reached and thread count. The other parameters are fixed, particularly NUM_TX = 100, probability = 50%, and threshold = 20. It can be seen that with an increase in the number of items, the total frequent sets, max level reached and thread count increases. Similar to the previous table, this is because more items per transaction gives better opportunities for more associations. Also, with an increase in the number of items, there is a proportionate increase in the number of threads launched. This interesting observation is not seen in other cases / variations.



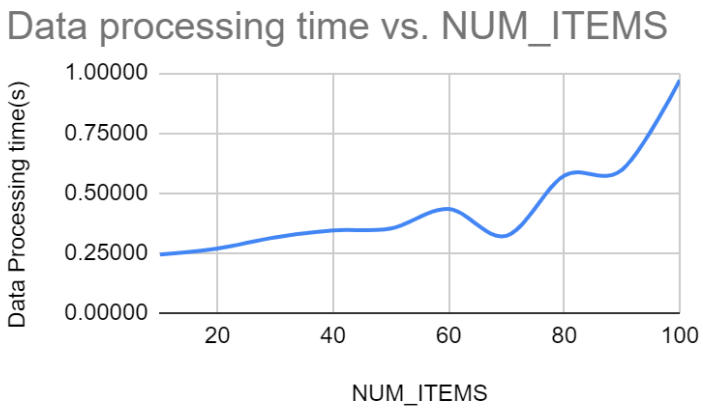
The above figure shows the variation in execution time as the number of items increases. It can be seen that the execution time increases with increase in NUM_ITEMS. Both multi core and GPU perform much better than single core. At NUM_TX = 100, GPU performs 10.8 times better over single core, while multicore performs 6.6 times better over single core. As NUM_ITEMS increase, a notable performance difference is observed between multicore and GPU. This is because more items generates more candidates which our CUDA kernel evaluates.

The next three figures discuss the profiling of CUDA implementation.





The above figure shows the variation in CPU Page faults and Data transfer as the number of items increases. The data transfer size doesn't reach GBs unlike previous cases. This is because the number of items are incremented by step size of 10 and not 50.



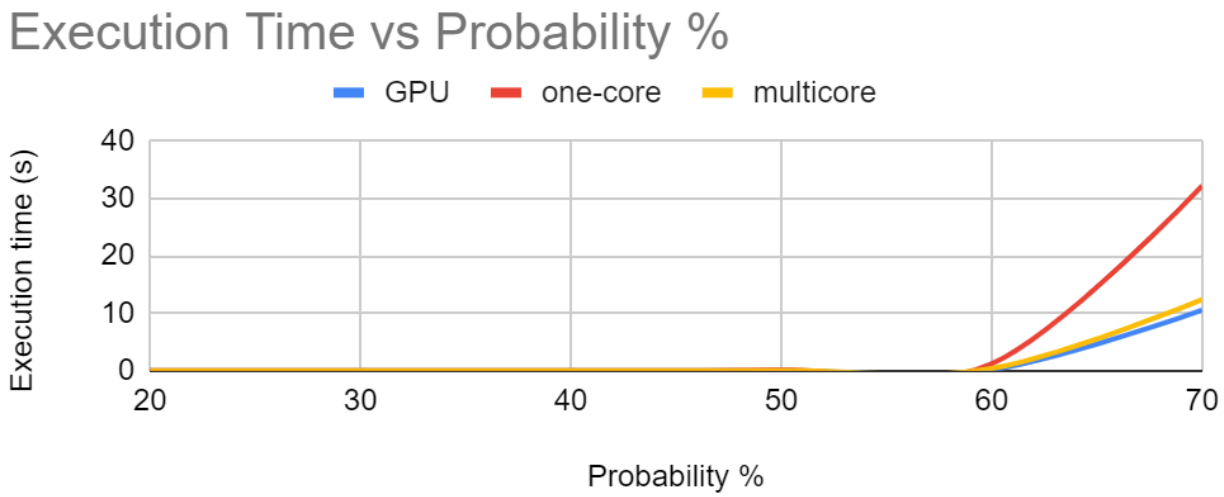
The above figure shows the variation in data processing time as the number of items increases. In the above three figures, the strange dips at 70 and 90 could be because of the distribution of the sample chosen. It could be corrected when the performed analysis is averaged over various samples of the same size.

5.4. Varying probability

Number of frequent itemsets and the highest level reached with varying probability-

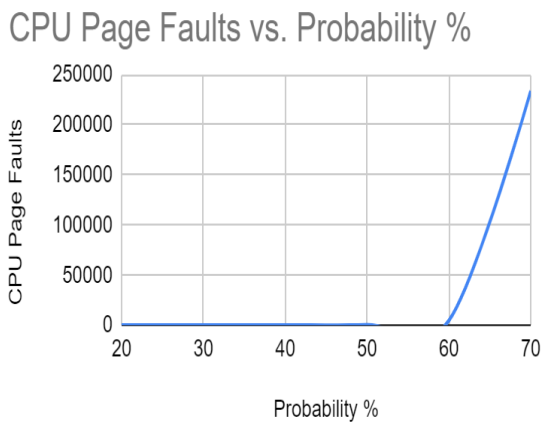
PROBABILITY	NUM_FREQ_ITEMSETS	MAX_LEVEL_REACHED
20%	23	L2
30%	49	L2
40%	312	L3
50%	1454	L5
60%	18251	L6
70%	699349	L9

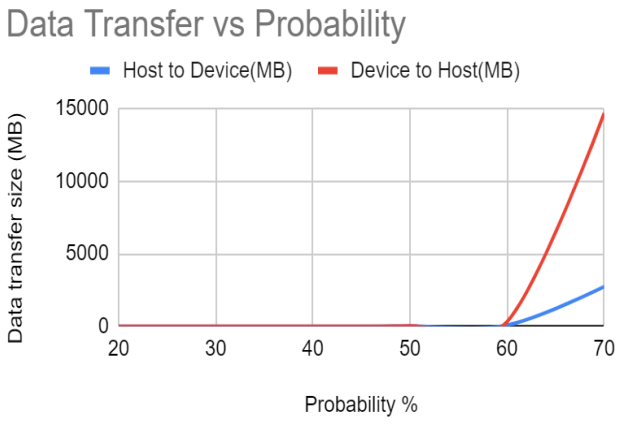
The above table shows the range of probability [0.2, 0.7] at a step size of 0.1 with the corresponding count of total frequent items sets and maximum level reached. The other parameters are fixed, particularly NUM_TX = 100, NUM_ITEMS = 50, and threshold = 20. It can be seen that with an increase in the probability, the total frequent sets and max level reached increases. Similar to the previous table, this is because a higher probability per transaction gives better opportunity for more associations.



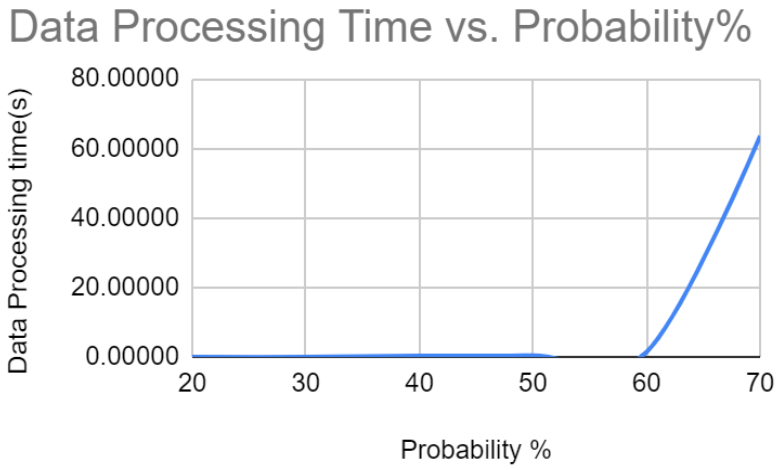
The above figure shows the variation in execution time as the % probability. It can be seen that the execution time exponentially increases. Both multi core and GPU perform much better than single core for a probability > 40%. At probability = 70%, GPU performs 3 times better over single core, while multicore performs 2.6 times better over single core. As the probability increases, the execution time of all implementations exponentially increases. This is because the computation complexity (association opportunity) increases with a higher value of probability.

The next three figures discuss the profiling of CUDA implementation.





The above figure shows the variation in CPU Page faults and Data transfer as the probability increases.



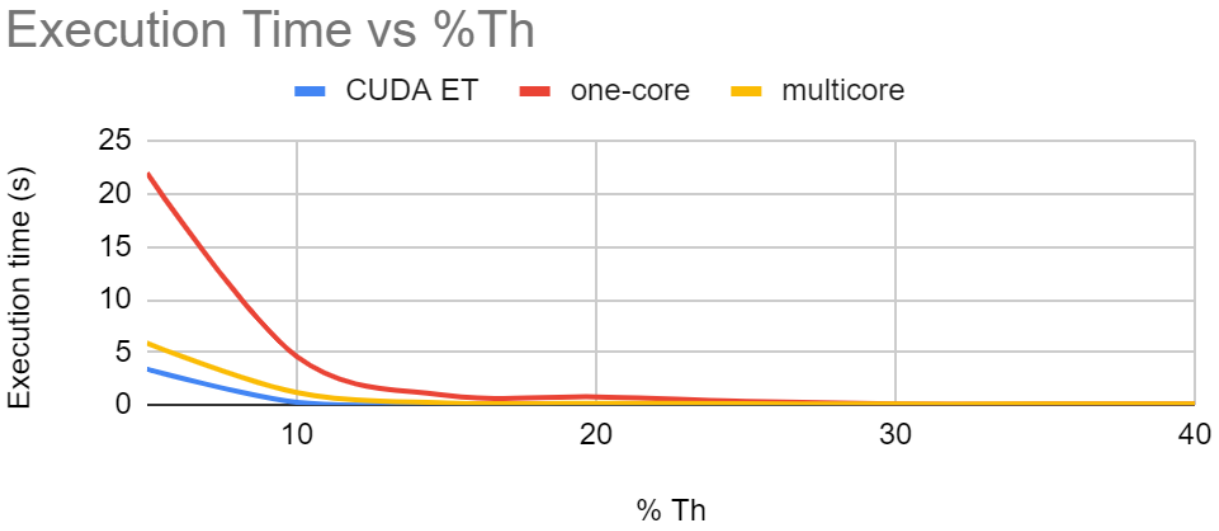
The above figure shows the variation in data processing time as the probability increases. In all the three figures, the value in the Y axis tends to exponentially increase after 60% probability.

5.5. Varying threshold

Number of frequent itemsets and the highest level reached with varying threshold-

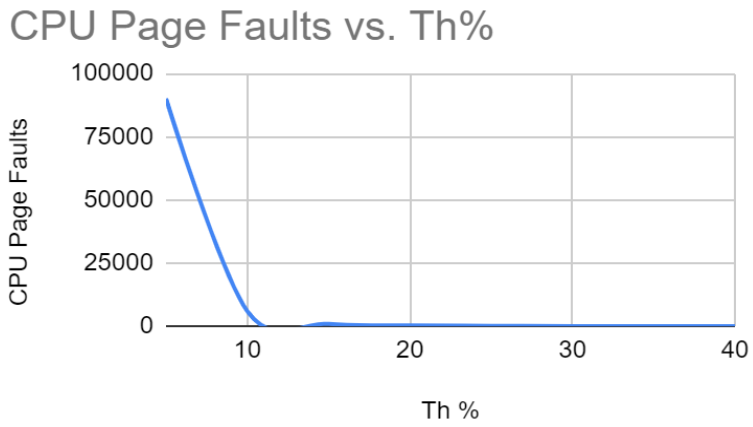
THRESHOLD	NUM_FREQ_ITEMSETS	MAX_LEVEL_REACHED
25	245898	L7
50	19749	L5
75	2429	L4
100	1251	L4
125	660	L3
150	61	L3
175	50	L2
200	50	L2

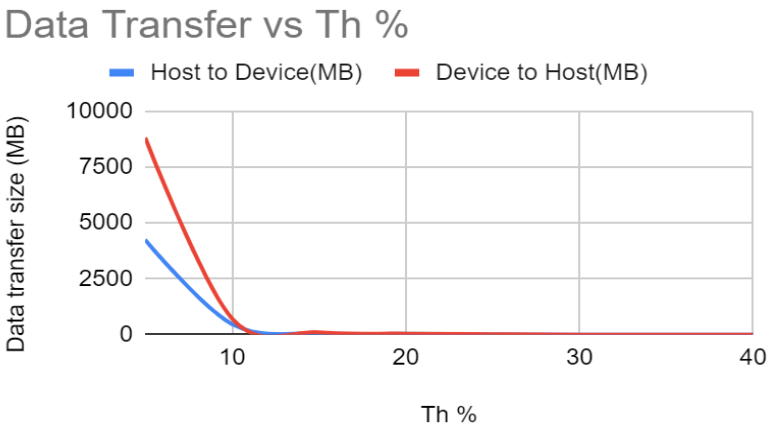
The above table shows the range of THRESHOLD [25, 200] at a step size of 25 with the corresponding count of total frequent items sets, maximum level reached and thread count. The other parameters are fixed, particularly NUM_TX = 500, NUM_ITEMS = 50, and probability = 50%. The values of threshold relates to % occurence in NUM_TX, which is [5% to 40%]. It can be seen that with an increase in the threshold, the total frequent item sets and max level reached decreases. This is because a higher value of threshold is analogous to a strong filter, which decreases the scope of association.



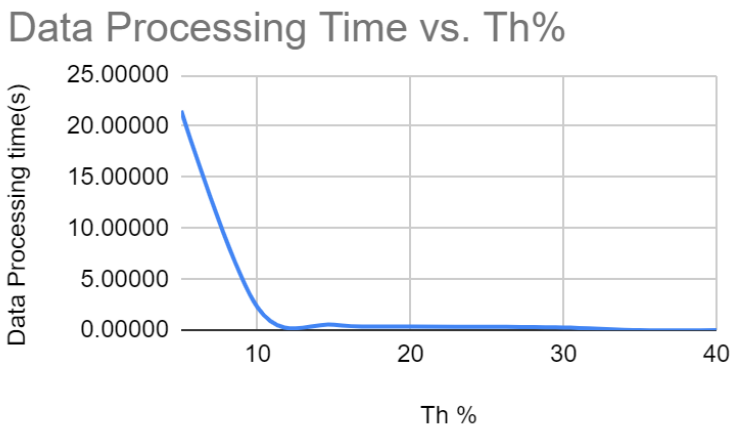
The above figure shows the variation in execution time as the % threshold increases. It can be seen that the execution time decreases with increase in threshold. Both multi core and GPU perform much better than single core for a lower value of threshold. At Threshold = 5%, GPU performs 6.5 times better over single core, while multicore performs 3.7 times better over single core. As the threshold increases, the execution time of all implementations converges. This is because the computation complexity (association opportunity) decreases with a higher value of threshold.

The next three figures discuss the profiling of CUDA implementation.





The above figures show the variation in CPU Page faults and Data transfer as the threshold increases.



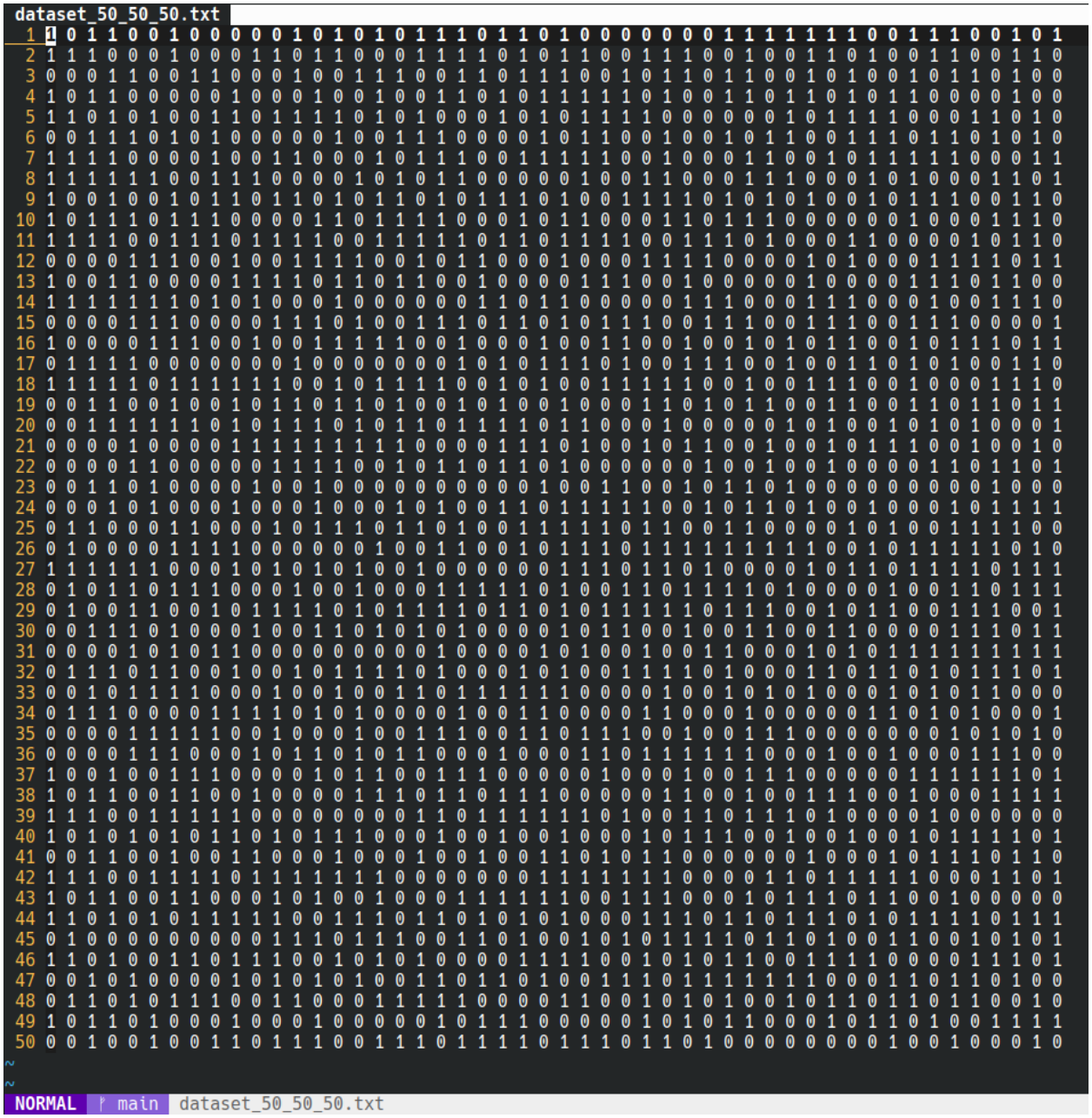
The above figure shows the variation in data processing time as the Threshold increases. In all the three figures, the value in the Y axis tends to exponentially decrease after the 12% threshold.

5.6. Special cases

In order to study how the skewness in data affects the results in comparison to a random distribution, 4 different variants with the following configuration were generated-

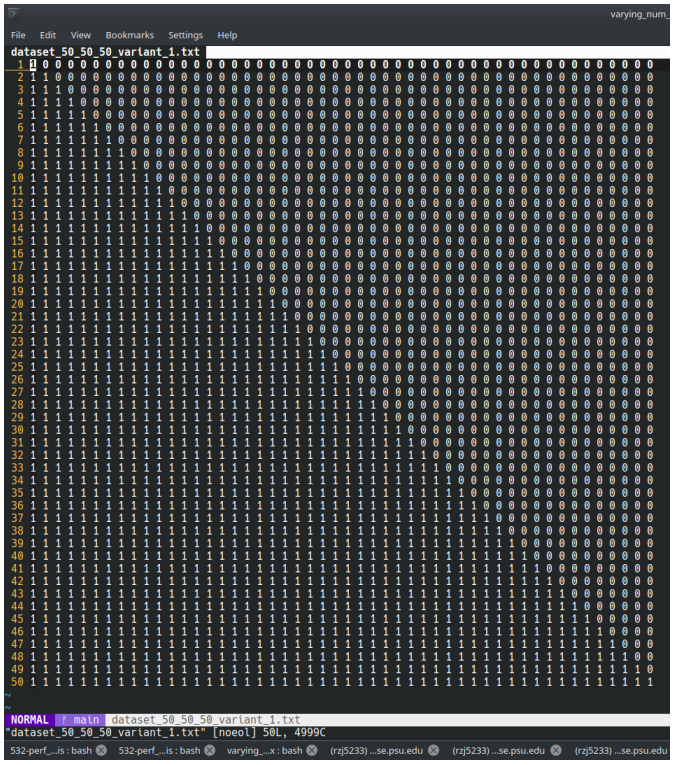
NUM_TX = 50
NUM_ITEMS = 50

The baseline randomly distributed dataset is as follows-

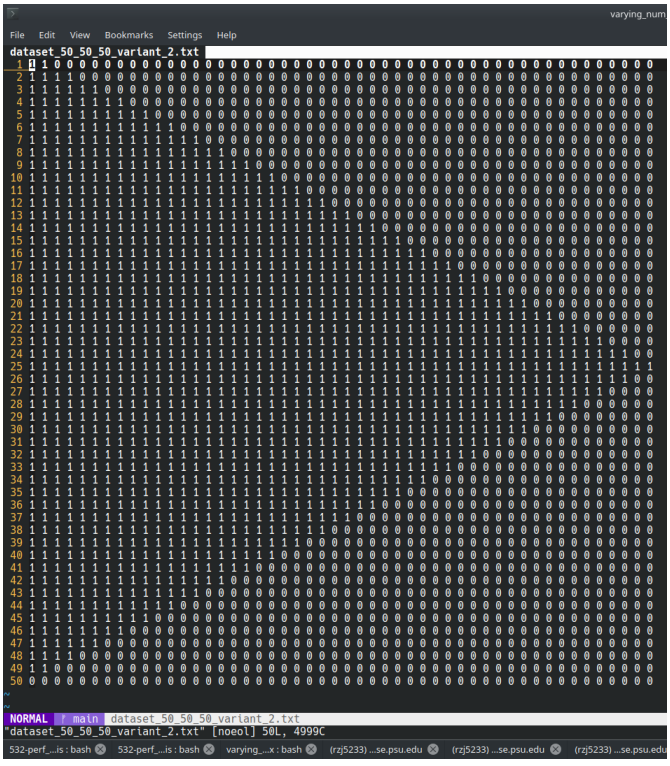


These 4 skewed datasets (variants) can be viewed as follows-

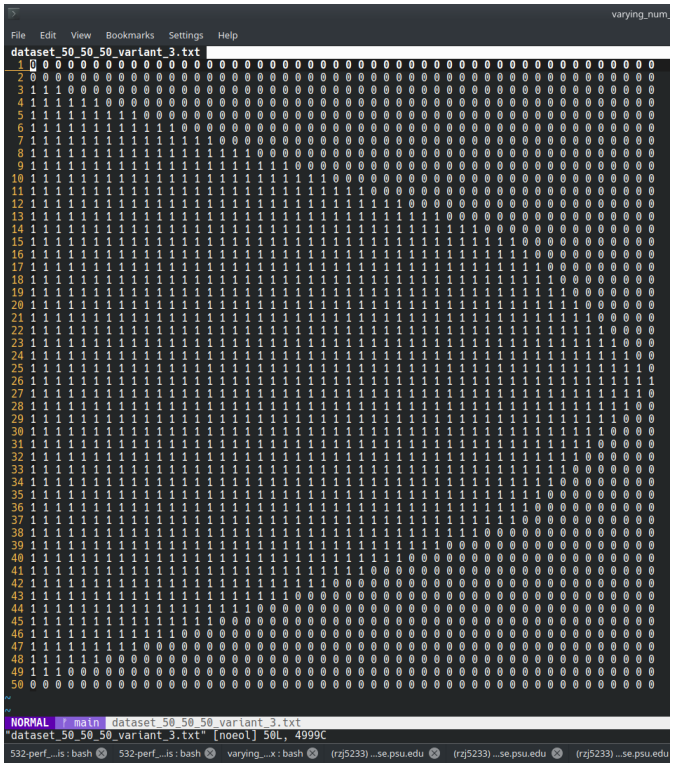
This variant is an increasing stepwise distribution. Item popularity in this distribution decreases from left to right i.e. the 1st item is the most popular since it occurs in all 50 transactions whereas the 25th item is less popular since it occurs in only 25 transactions.



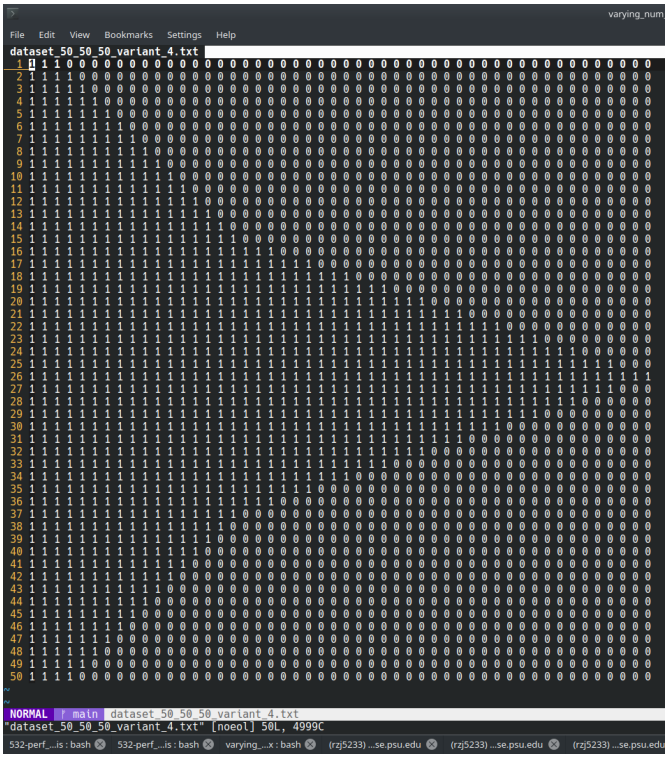
This variant is an increasing and then decreasing stepwise distribution. Item popularity in this case too, decreases from left to right.



This variant is analogous to a gaussian / normal distribution. Item popularity again decreases from left to right, the leftmost item being the most popular.



This variant is analogous to an exponential distribution. Item popularity again decreases from left to right, the rightmost item being the least popular.

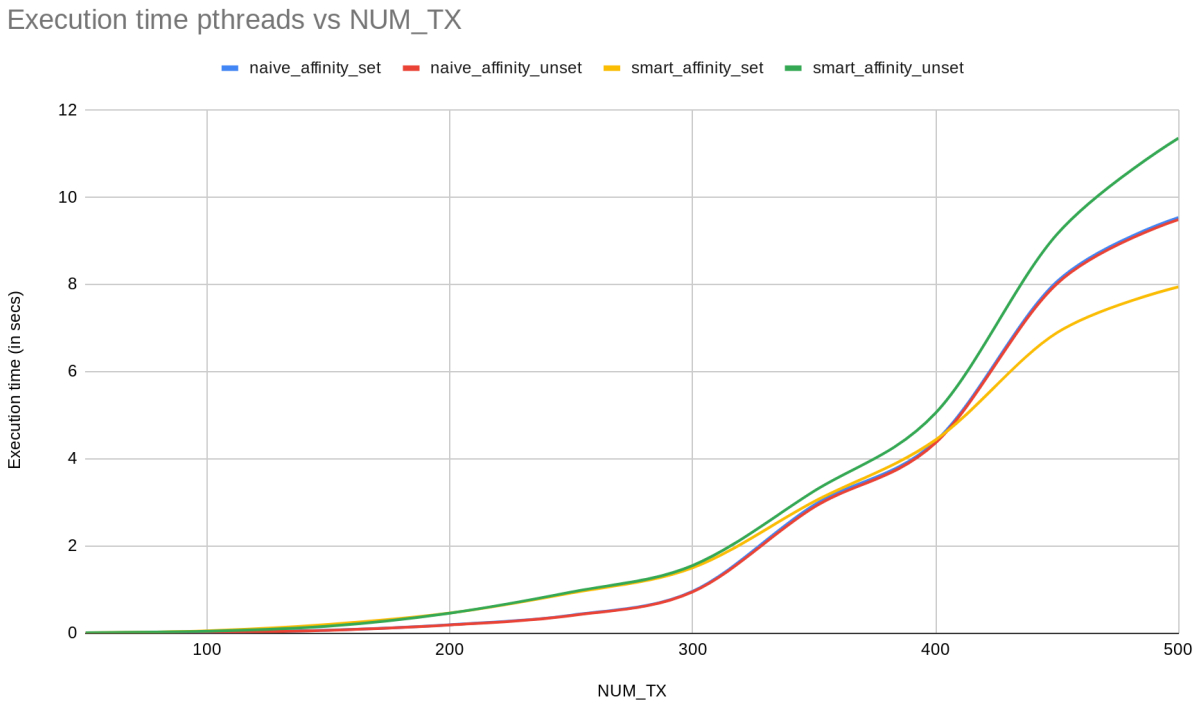


Using these distributions, an attempt was made to analyze how skewness in item popularity affects the required computational effort.

5.7. Summarizing the comparisons between different cases

Varying NUM_TX with cpu affinity unset and set (comparison between the naive and smart implementations)-

In case of the pthreads implementation, upon calculating the number of threads to be launched based on the number of equivalence classes at level 2 and determining the number of available cores, cpu affinity for each thread is set using a round robin mechanism.



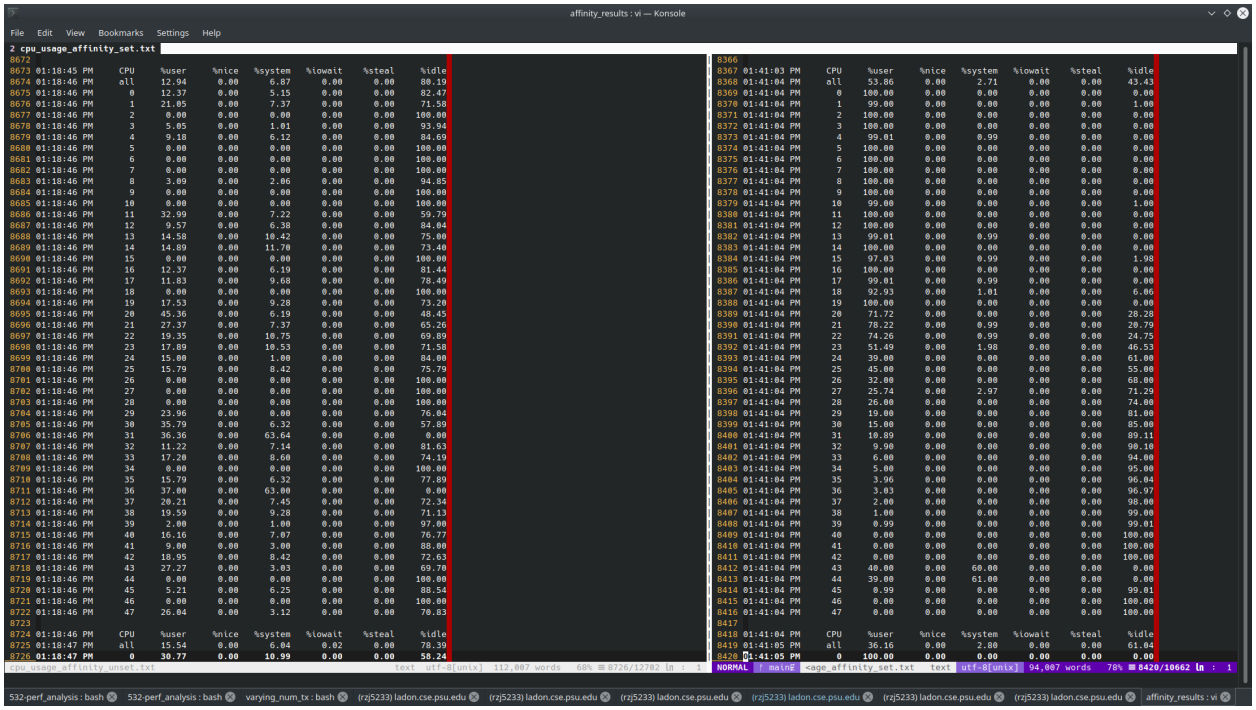
As seen in the graph above, there is almost no difference in terms of the naive implementation when core affinity is set or unset. In terms of the smart implementation, setting the cpu affinity helps since the execution time reduces for higher transaction sizes as compared to when the cpu affinity is unset. Also, as can be observed here, there comes a crossover point after which the smart implementation performs better than the naive implementation. Before this crossover point, the cost of maintaining the extra txids data structure exceeds that of exhaustively searching through the entire dataset.

The sar command line utility was used to record the cpu usage characteristics (it is similar to top and htop and reports the same statistics).

```
sar -P ALL <interval> <num_of_times> > <output_file>
```

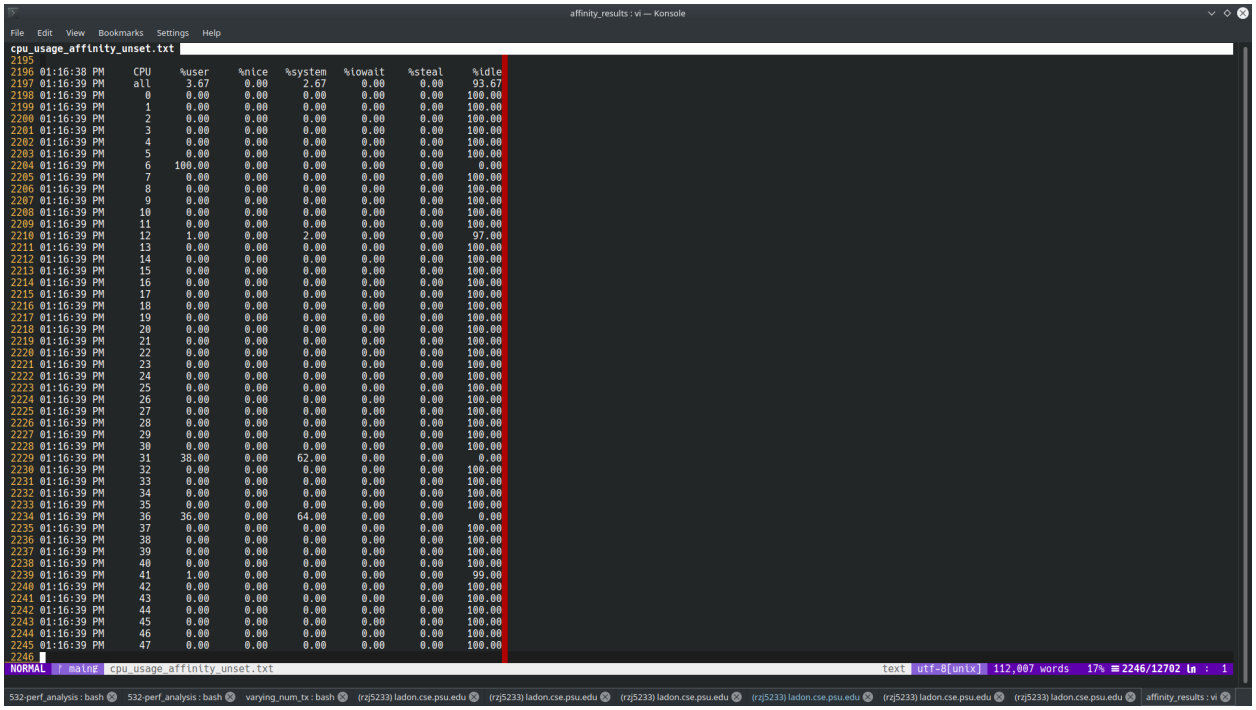
Example: `sar -P ALL 1 500 > <output_file>`

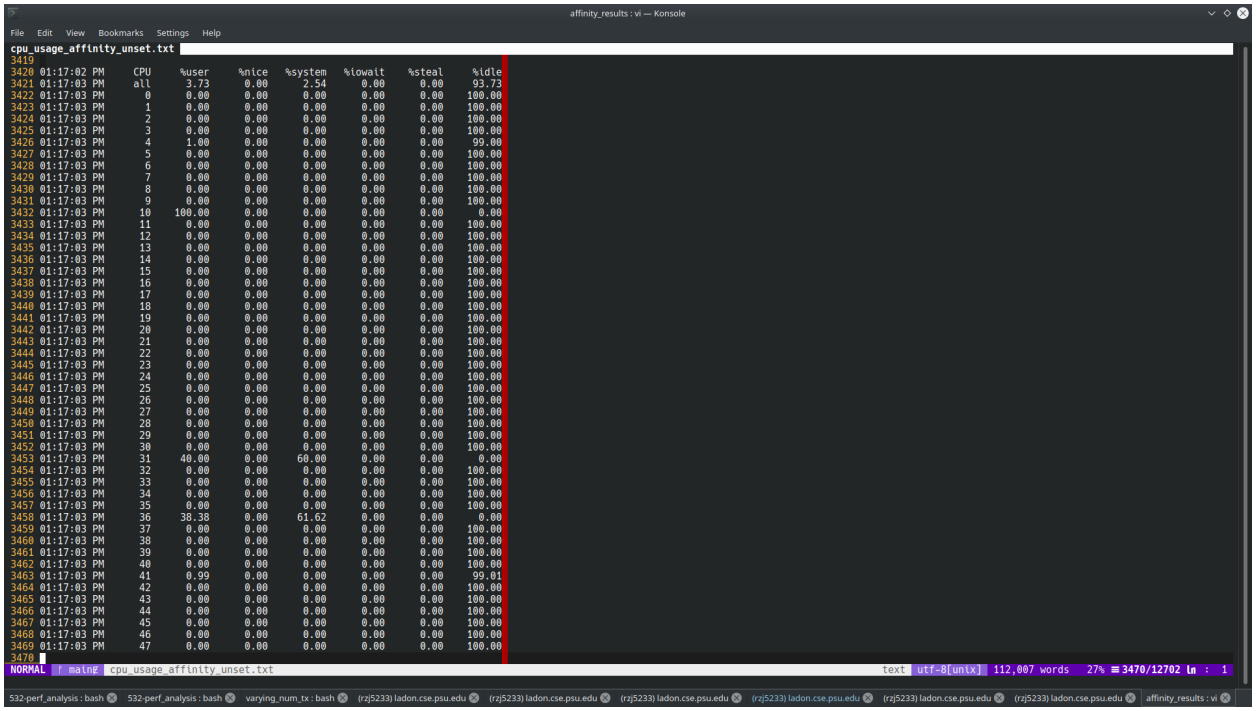
CPU usage pattern with cpu affinity unset vs set-



The above data shows how the cpu usage has been monitored. One observation here is that when the cpu affinity is not set, the % utilization of the cores in use is not significantly high, whereas when it is set, is it high for cores that are in use.

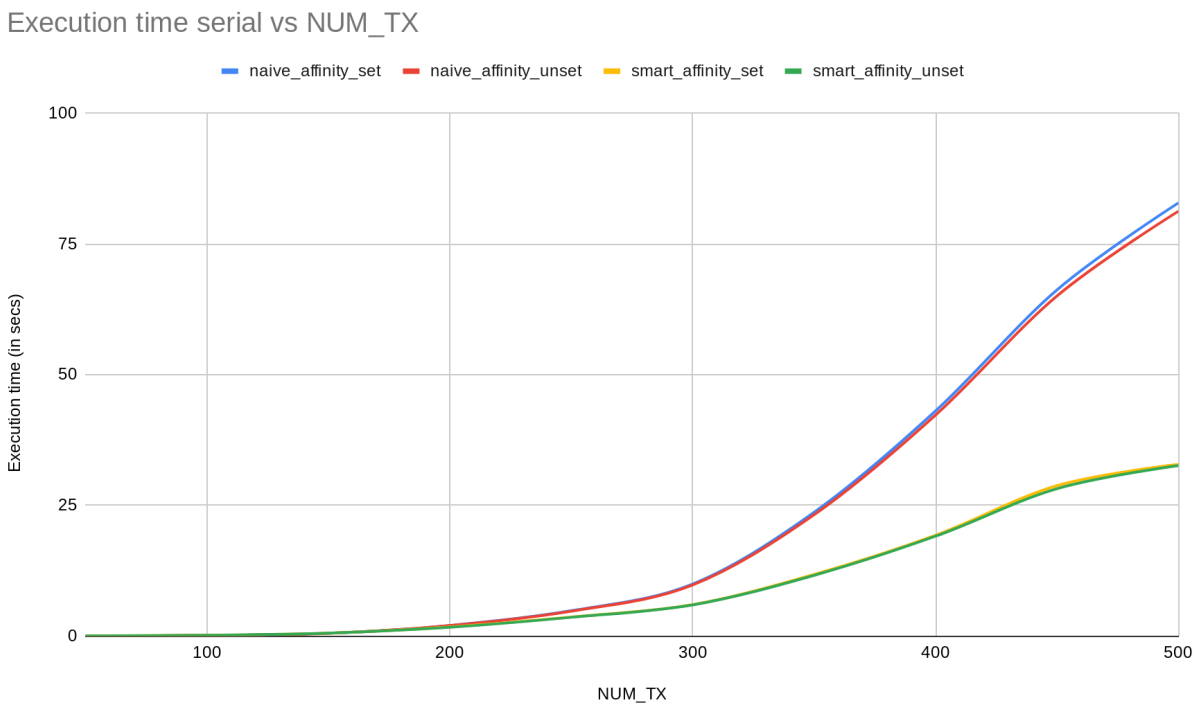
In case of the serial implementation, sar shows that at a time, only one core is utilized for the computation (though, this core might not be the same throughout the entire computation, since as seen from the sar output, the main thread seems to migrate between cores)-



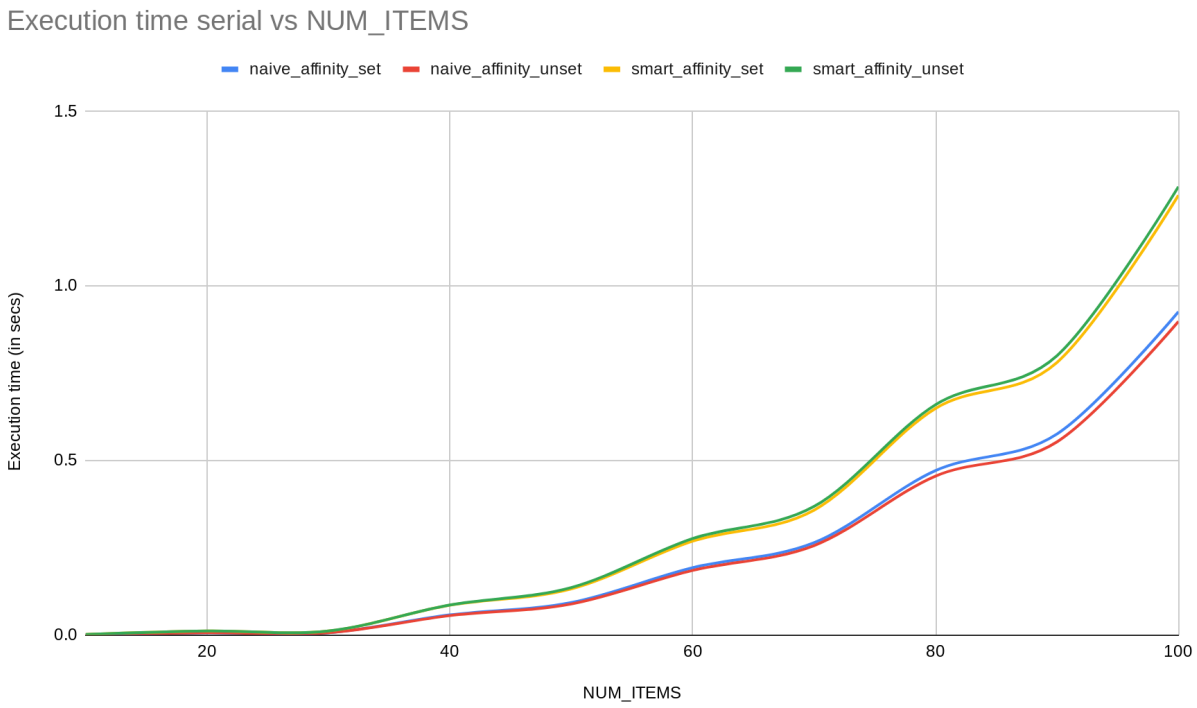


Comparison of naive and smart serial implementations-

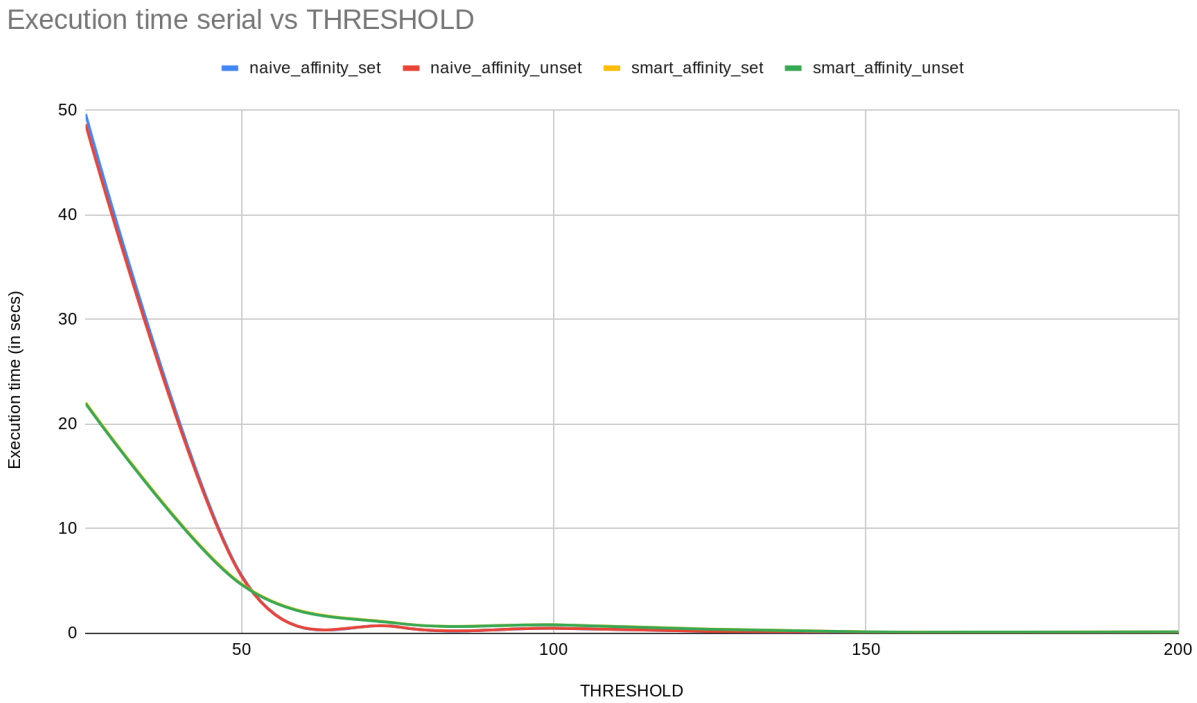
Varying number of transactions



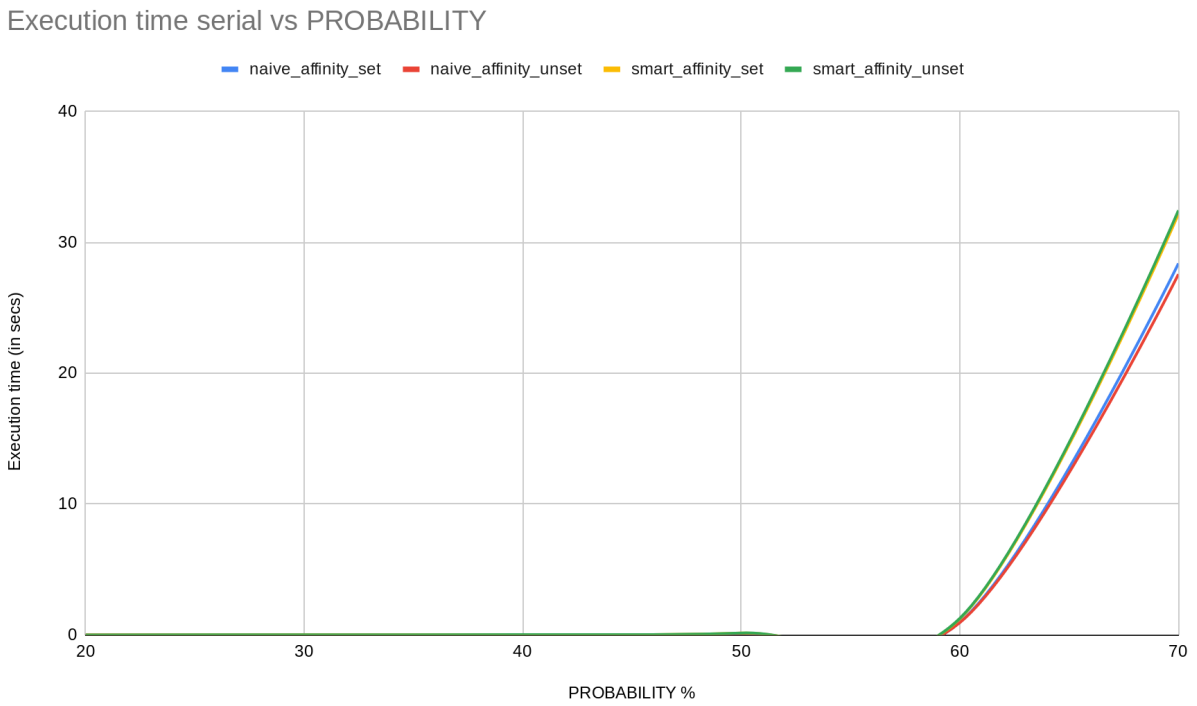
Varying number of items



Varying threshold



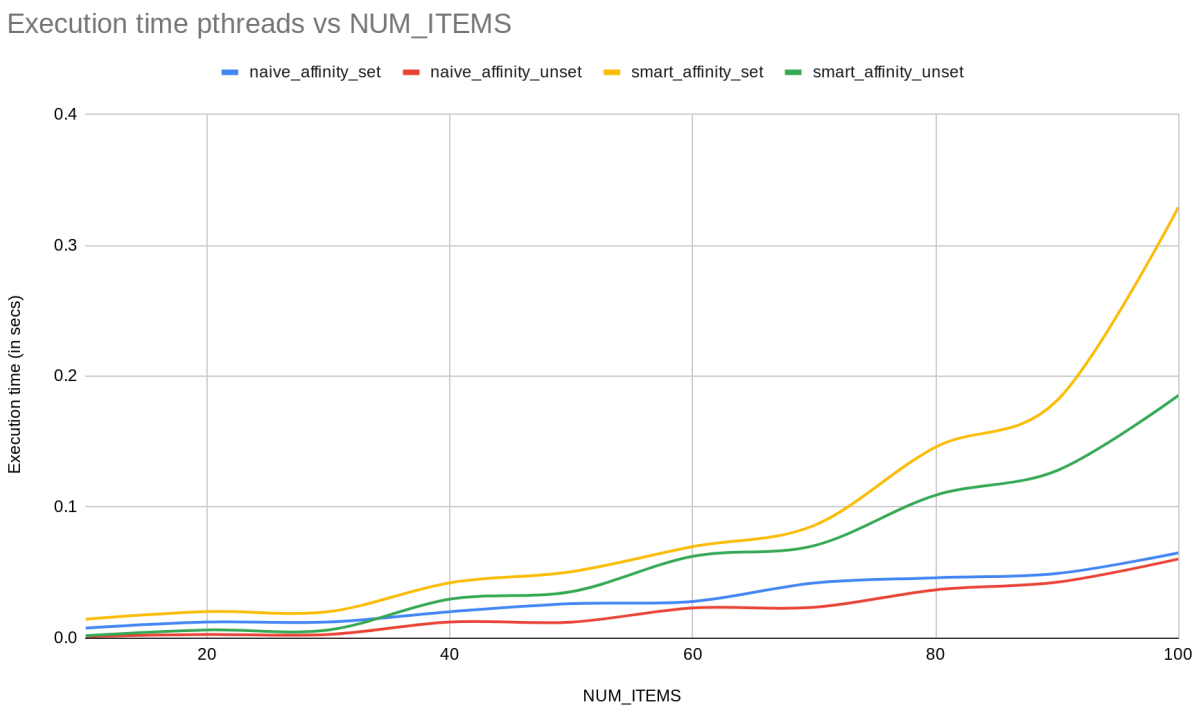
Varying probability



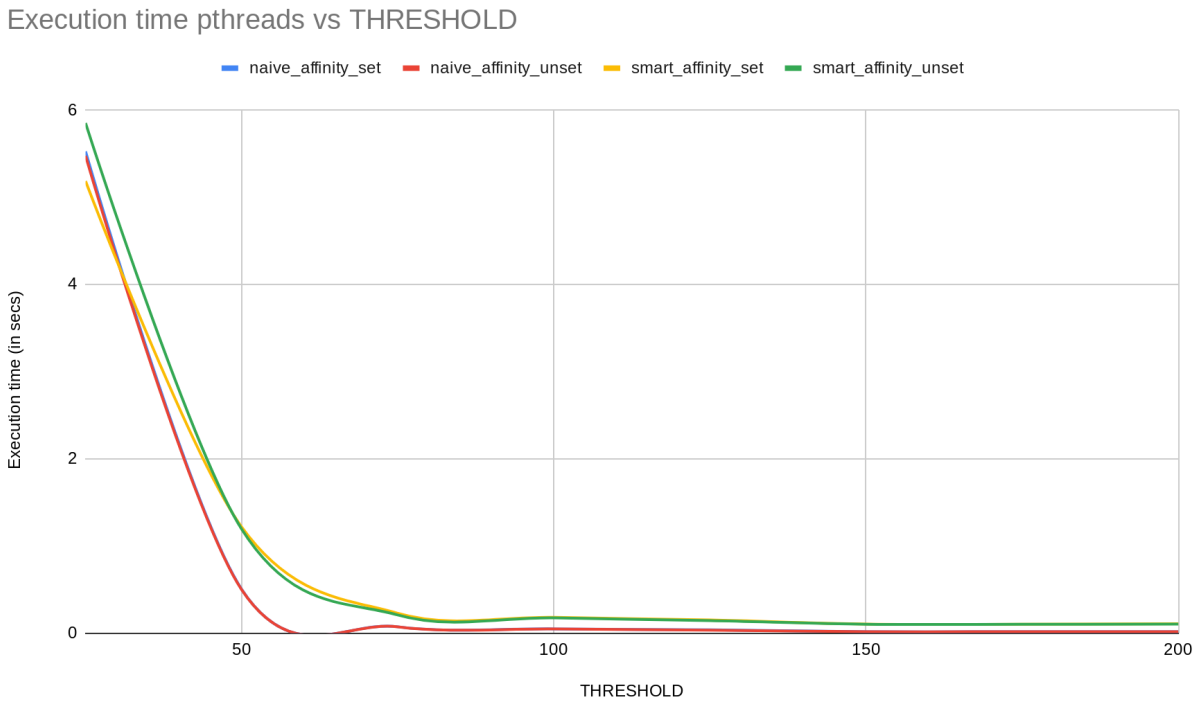
Comparison of naive and smart pthreads implementations-

Note: For a varying number of transactions, the graph has already been explained before.

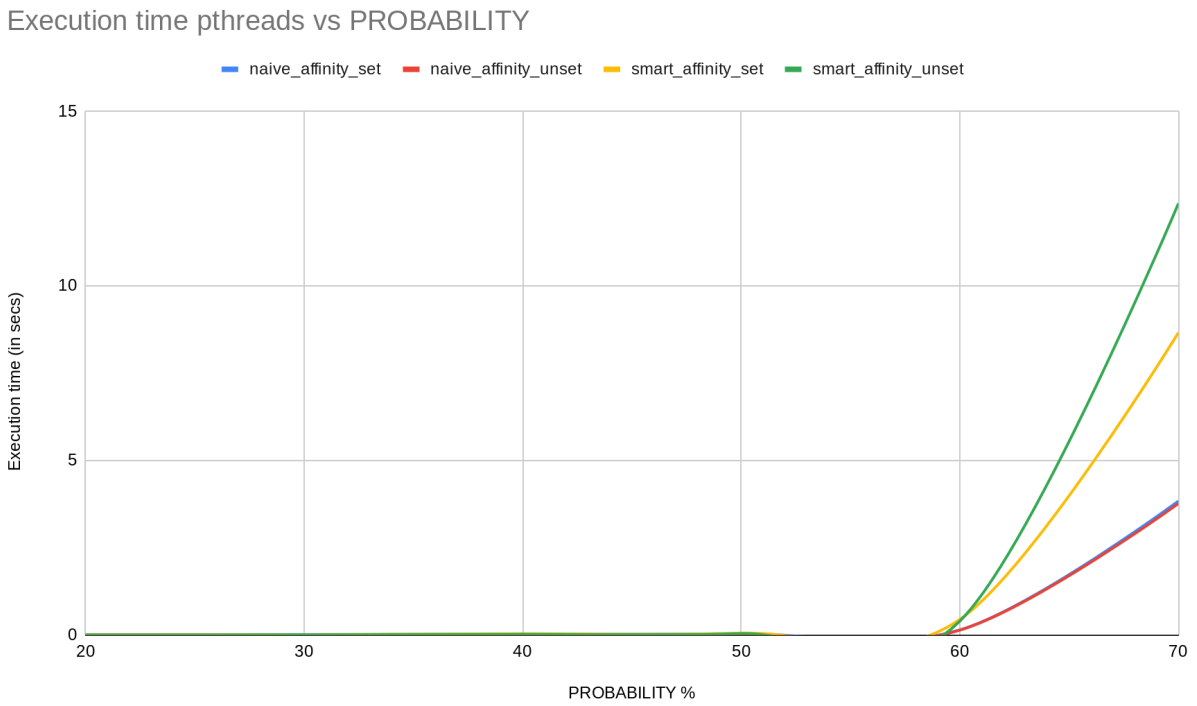
Varying number of items



Varying threshold

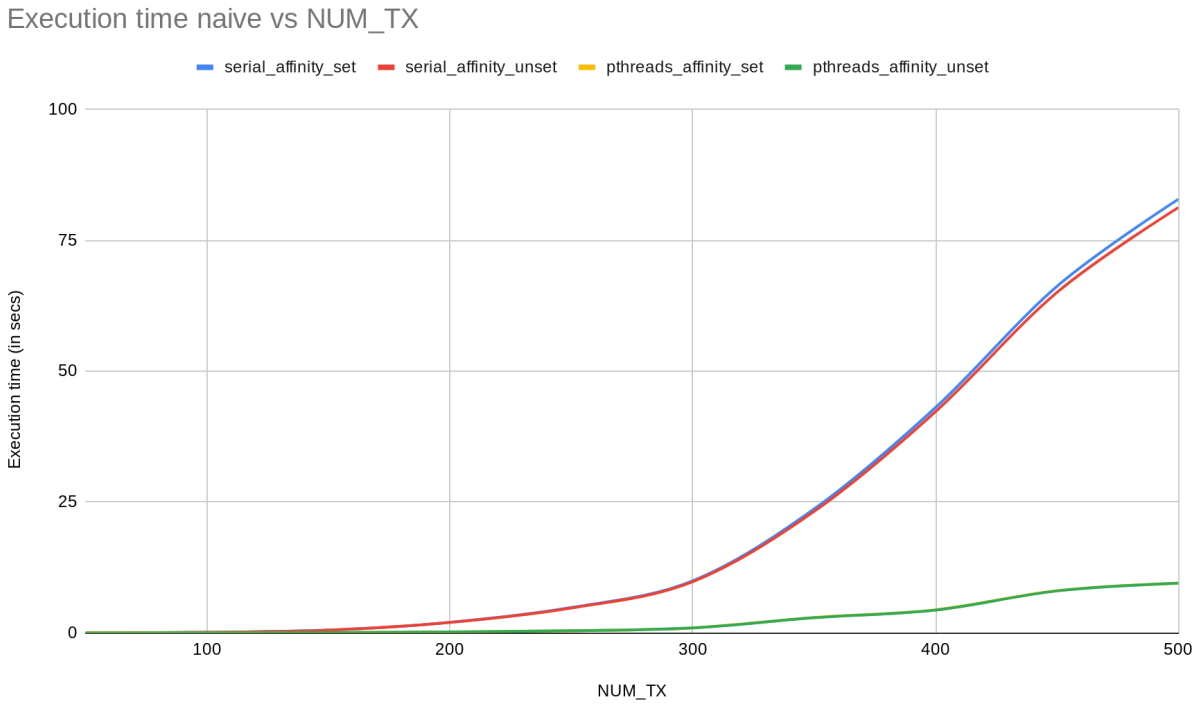


Varying probability

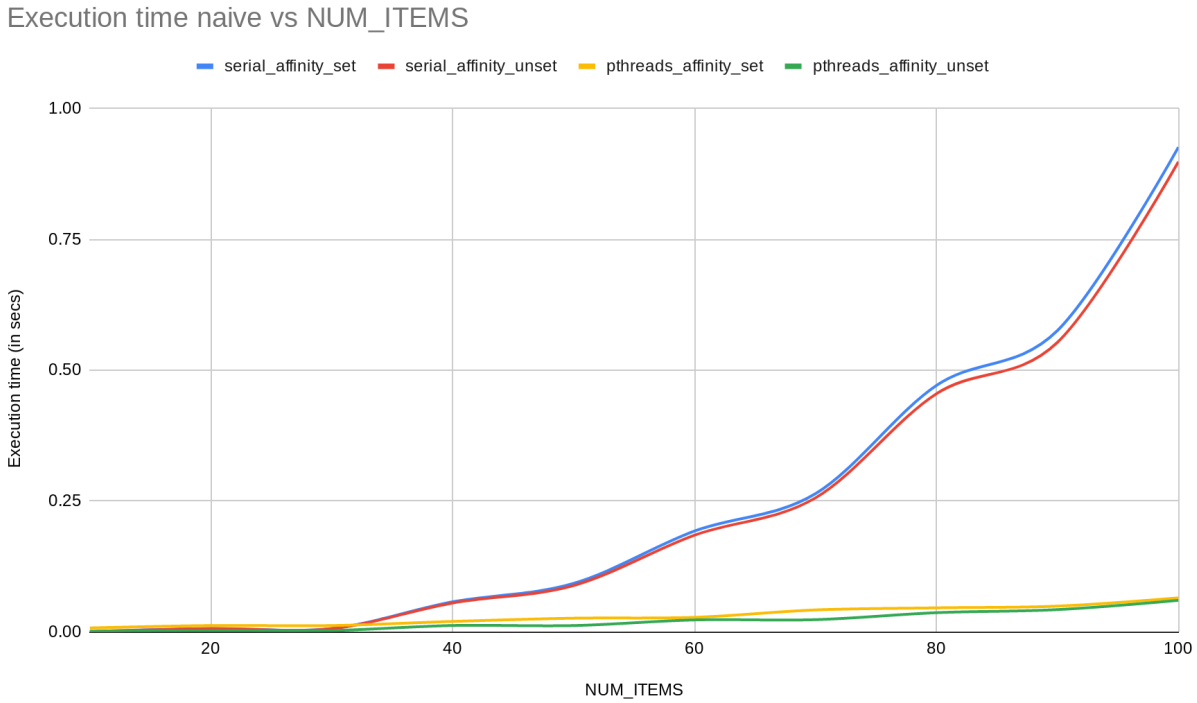


Comparison of naive serial and pthreads implementations-

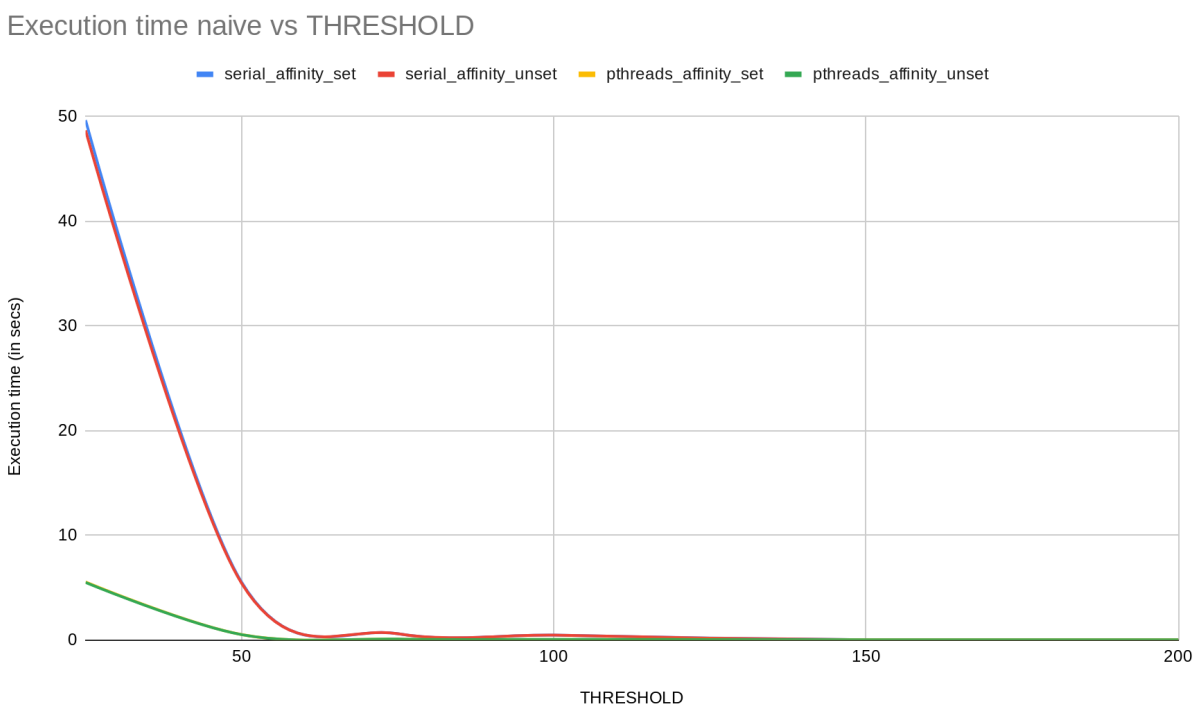
Varying number of transactions



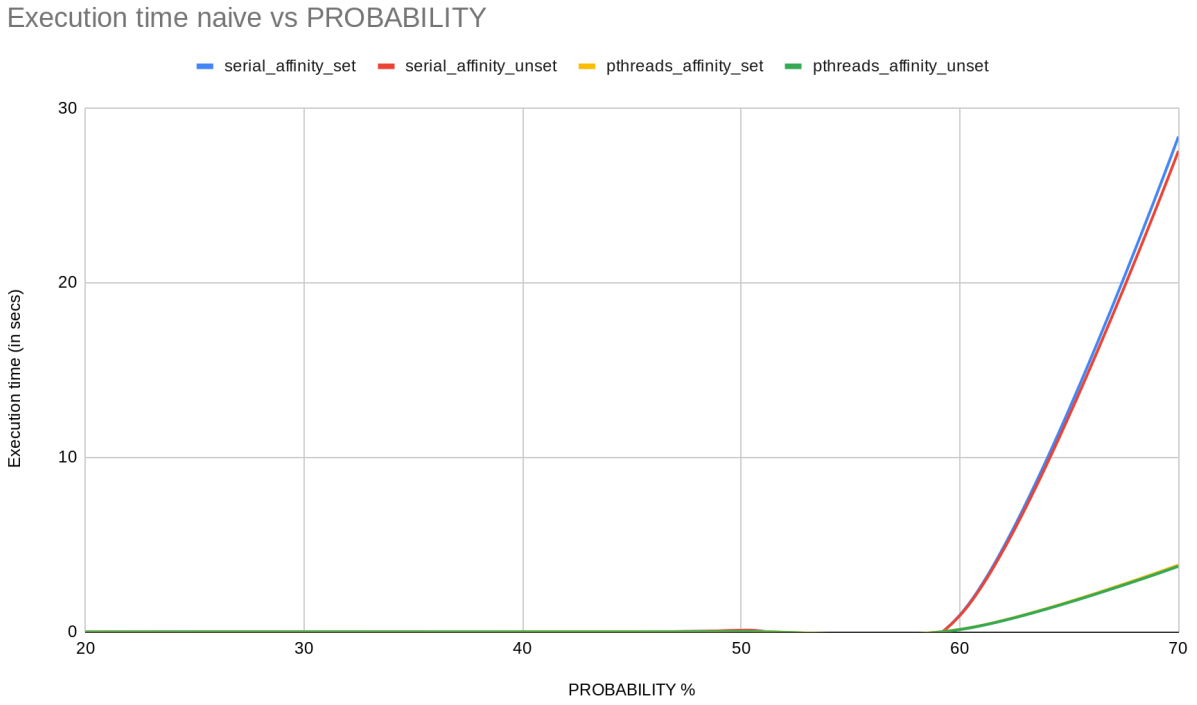
Varying number of items



Varying threshold

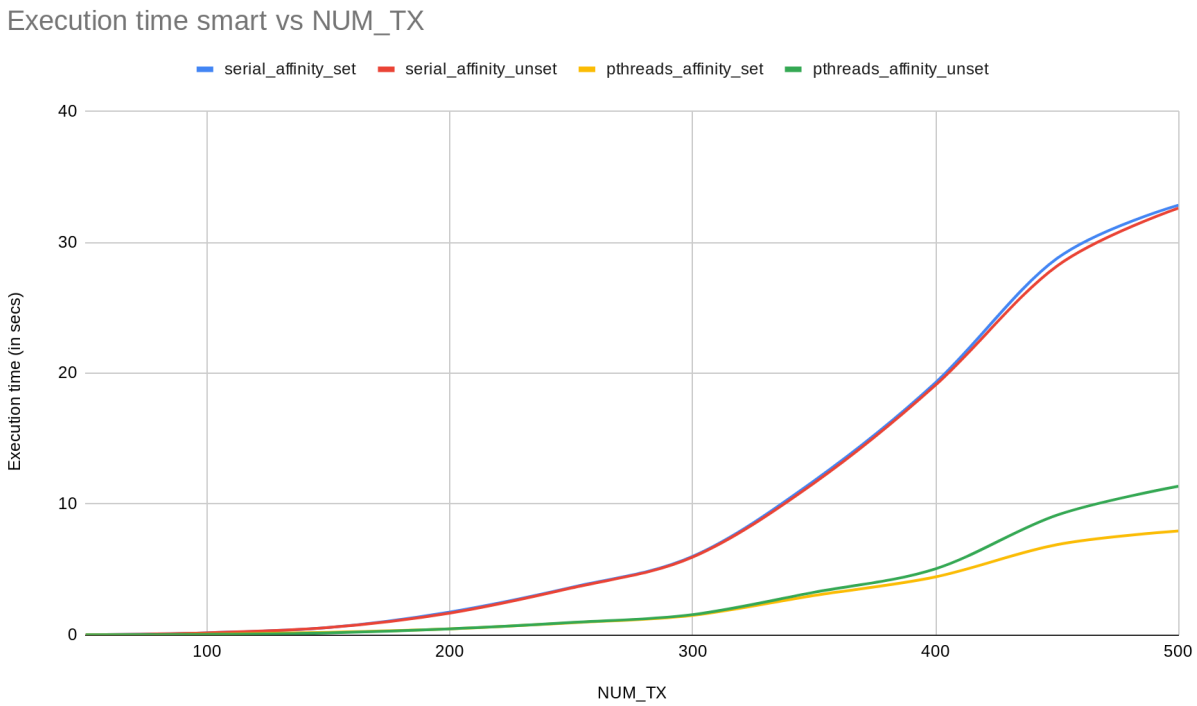


Varying probability

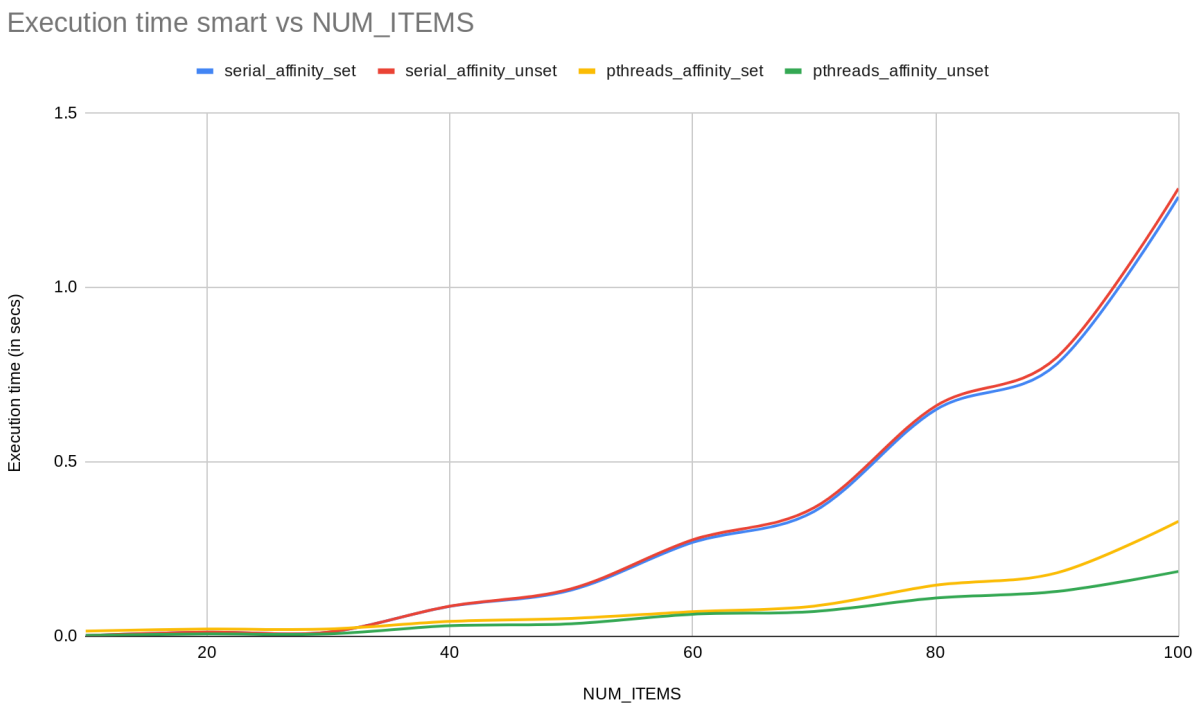


Comparison of smart serial and pthreads implementations-

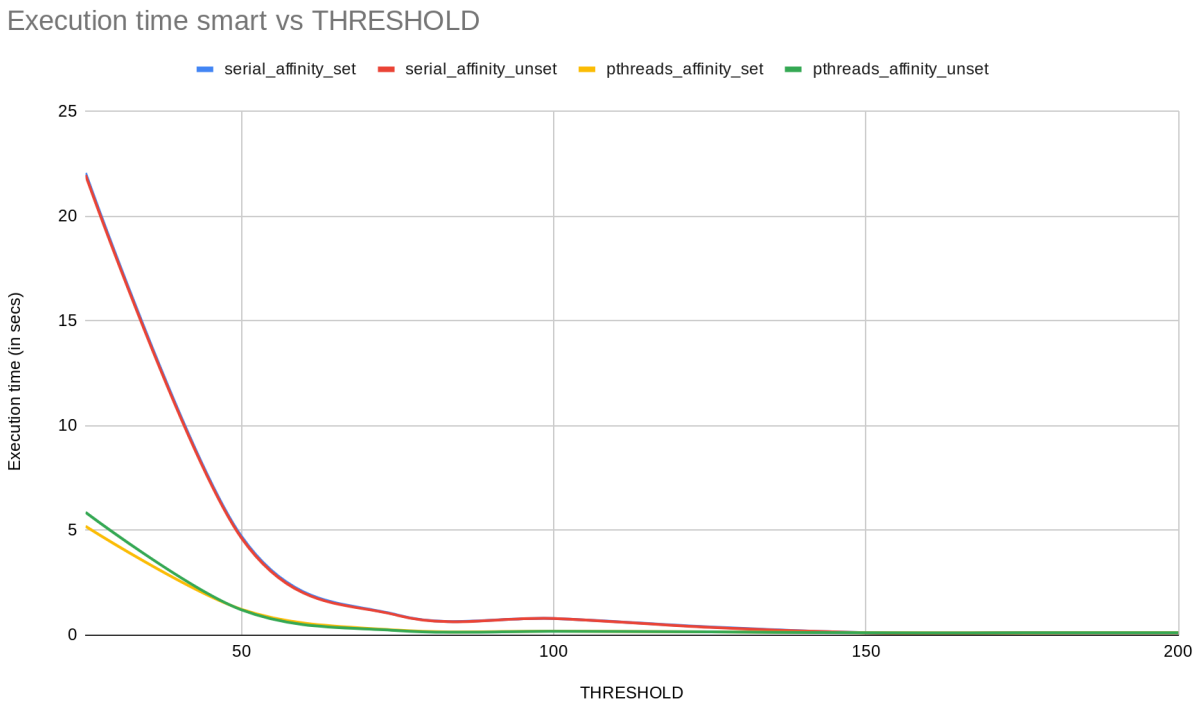
Varying number of transactions



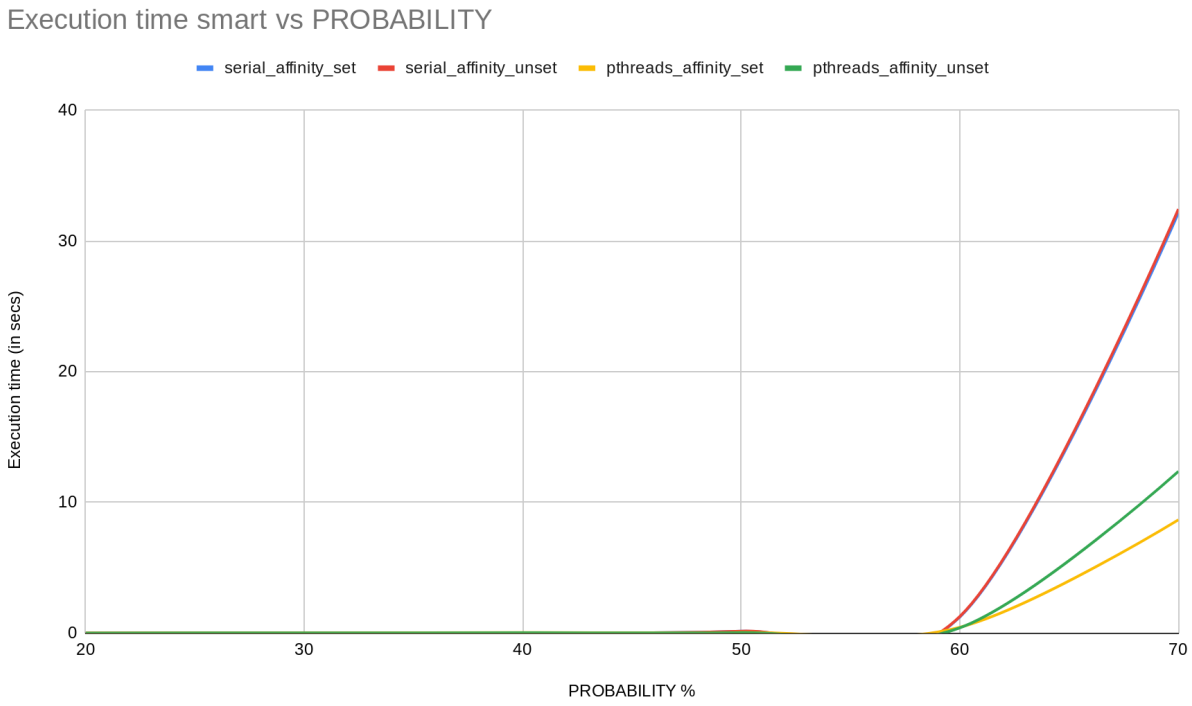
Varying number of items



Varying threshold



Varying probability



As observed in all the graphs above, the pthreads implementation performs significantly better than the serial implementation, irrespective of whether cpu affinity is set or unset. Further, there is no significant difference with respect to core affinity being set or not. Finally, in most of the above cases except in case of varying transactions for the serial implementation (smart performs better in this case), naive seems to perform better than smart (with a few minor anomalies in some cases, execution time vs threshold, for example). A possible reason for this has been explained in the report above (see the section of cpu affinity set and unset), though further analysis is required to get a complete understanding of this scenario.

Results obtained for higher number of transactions-

NUM_ITEMS = 50
THRESHOLD = 20
PROBABILITY = 0.5

num_tx	exec_time (in secs)							
	naive_serial		smart_serial		naive_pthreads		smart_pthreads	
	affinity_set	affinity_unset	affinity_set	affinity_unset	affinity_set	affinity_unset	affinity_set	affinity_unset
500	82.891433	81.302133	32.852633	32.628367	9.539257	9.493583	7.94909	11.363633
750	500.224	496.788	128.441	129.754	55.8848	55.9105	62.2076	61.7823

1000	1286.22	1257.01	279.091	279.9	149.273	147.895	111.15	170.568

The following tabulated sequential measurement of time (the last 4 columns in tables below) is not an introduction of some new algorithm or method to perform the task at hand, but is just a measurement of how much time would be needed if all the threads' work is aggregated in a sequential manner instead of having any time overlap which threading actually provides.

Varying number of transactions

num_tx	exec_time (in secs)							
	naive_pthreads		smart_pthreads		naive_sequential_pthreads		smart_sequential_pthreads	
	affinity_set	affinity_unset	affinity_set	affinity_unset	affinity_set	affinity_unset	affinity_set	affinity_unset
50	0.00234	0.001781	0.008364	0.007621	0.000138	0.000195	0.007085	0.007427
100	0.022403	0.014834	0.051744	0.040673	0.109489	0.119017	0.278568	0.196157
150	0.065483	0.062736	0.197969	0.157017	0.608572	0.59409	2.669247	1.48227
200	0.1907	0.18525	0.458289	0.456888	2.207017	2.118227	7.721343	6.737537
250	0.409843	0.401534	0.91913	0.942431	5.054997	4.971323	15.632733	14.5601
300	0.961019	0.946423	1.49761	1.551983	10.259833	10.1094	23.7267	23.501667
350	2.942077	2.89004	3.018837	3.259337	25.7541	24.681133	44.429233	46.913333
400	4.405237	4.367143	4.434817	5.05253	44.963267	44.122867	63.884833	71.8936
450	8.07115	8.02166	6.896717	9.15751	68.399167	67.3581	88.599667	114.595667
500	9.539257	9.493583	7.94909	11.363633	85.0115	84.053933	105.581333	144.528333

Varying number of items

num_items	exec_time (in secs)			
	naive_pthreads	smart_pthreads	naive_sequential_pthreads	smart_sequential_pthreads

	affinity_ set	affinity_ unset	affinity_ set	affinity_ unset	affinity_ set	affinity_ unset	affinity_ set	affinity_ unset
10	0.007502	0.00119	0.014263	0.001721	0.000583	0.000641	0.001773	0.001621
20	0.012103	0.002637	0.020054	0.006084	0.006174	0.007258	0.01457	0.015081
30	0.023424	0.004747	0.034472	0.014736	0.021077	0.022852	0.051466	0.050786
40	0.019995	0.012194	0.042138	0.02959	0.056462	0.06043	0.151517	0.10602
50	0.02619	0.012056	0.050613	0.035323	0.09157	0.100593	0.223726	0.169966
60	0.027828	0.022885	0.069786	0.062365	0.199794	0.212304	0.412951	0.356435
70	0.041963	0.023368	0.085939	0.070525	0.297285	0.282892	0.673827	0.483343
80	0.045959	0.036711	0.145888	0.109074	0.529955	0.50452	1.86832	0.984555
90	0.049237	0.042574	0.181462	0.127912	0.64428	0.612966	2.354497	1.156083
100	0.065052	0.060287	0.329298	0.185341	1.128857	1.077317	5.986547	2.222963

Varying threshold

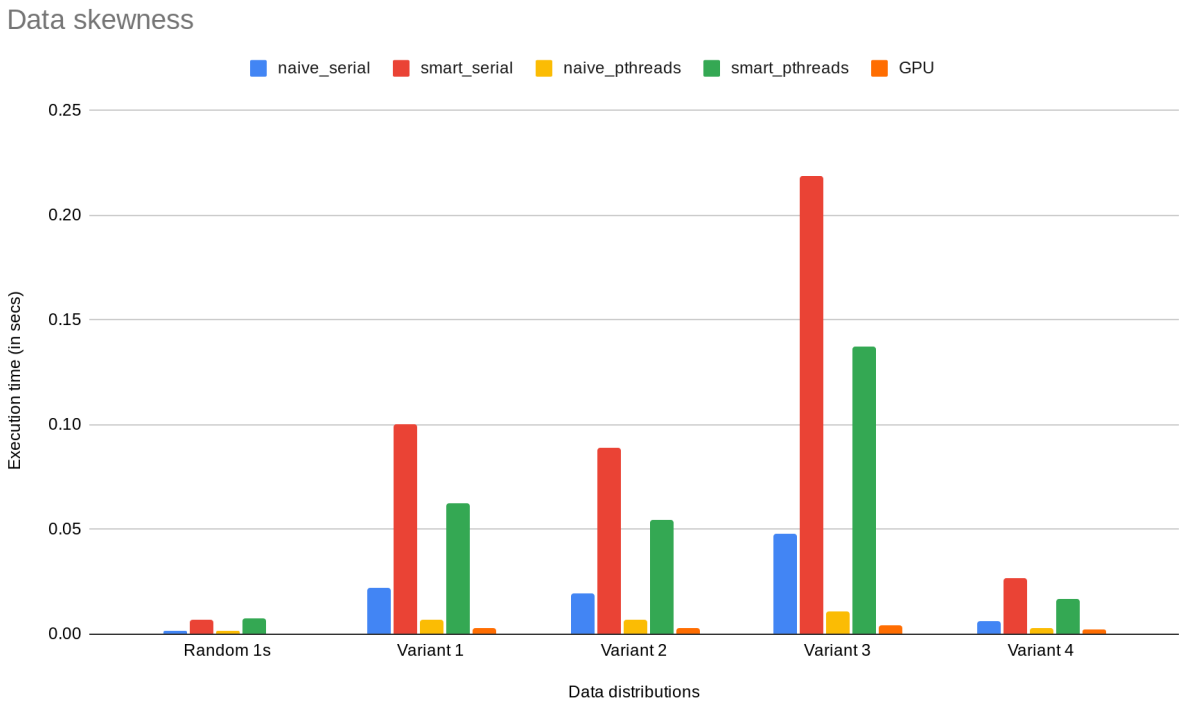
threshold	exec_time (in secs)							
	naive_pthreads		smart_pthreads		naive_sequential_pthreads		smart_sequential_pthreads	
	affinity_ set	affinity_ unset	affinity_ set	affinity_ unset	affinity_ set	affinity_ unset	affinity_ set	affinity_ unset
25	5.529357	5.479483	5.189437	5.85543	51.7116	50.779933	75.606967	84.7291
50	0.501456	0.496973	1.21638	1.192073	5.68478	5.633617	19.6284	17.111933
75	0.071761	0.071238	0.225797	0.207486	0.620428	0.610308	1.561743	1.089
100	0.047836	0.046896	0.180064	0.173473	0.471179	0.464163	0.858971	0.856926
125	0.036768	0.033892	0.149795	0.142819	0.165763	0.164941	0.417571	0.388114
150	0.01587	0.01533	0.102922	0.101526	0.000767	0.000813	0.102478	0.101515
175	0.014782	0.014736	0.100035	0.100693	0.000316	0.000323	0.100035	0.100693
200	0.014792	0.01474	0.105695	0.101807	0.000316	0.000321	0.105695	0.101807

Varying probability

probability	exec_time (in secs)							
	naive_pthreads		smart_pthreads		naive_sequential_pthreads		smart_sequential_pthreads	
	affinity_set	affinity_unset	affinity_set	affinity_unset	affinity_set	affinity_unset	affinity_set	affinity_unset
20%	0.000466	0.000471	0.000992	0.001033	0.000064	0.000055	0.000992	0.001033
30%	0.00214	0.002103	0.006143	0.006246	0.000065	0.000063	0.006143	0.006246
40%	0.018938	0.005166	0.031933	0.015904	0.012099	0.00802	0.026642	0.02414
50%	0.030355	0.013207	0.051678	0.0371	0.096459	0.107021	0.177514	0.176375
60%	0.144184	0.134687	0.441738	0.395971	1.094197	1.017757	6.563383	4.75761
70%	3.834503	3.76301	8.667087	12.372633	30.447267	30.1052	99.315233	144.856

Special datasets (skewness of item popularity)-

For analysis of all the datasets mentioned in Section 5.6., the threshold has been set to 20 (same as the baseline config), whereas the level has been set to 3 since the baseline results only generate up to a max level of 3.



The main observation here is that for all the variant datasets, the execution times are considerably higher than that of the baseline dataset. Along with this, there is a significant difference even in the execution times between different variant datasets. This indicates that the skewness pattern in the data also affects the required computation for the same workload size i.e. the distribution of data does matter. The smart implementation performing worse than the naive one is due to the reason mentioned previously (see the cpu affinity set and unset results). Though, again as expected, pthreads implementation performs better than the serial one.

Another interesting observation is that GPU performs much better compared to other implementations in all the skewness variants. This could be because all these variants support data locality in memory accesses as evident from the closeness of 1's as seen in the figures in Section 5.6.

The count of 1s in each of the compared datasets is as shown below-

Dataset	Count (out of total)
Baseline	1238 / 2500
Variant 1	1275 / 2500
Variant 2	1250 / 2500
Variant 3	1486 / 2500
Variant 4	1024 / 2500

For each of these datasets, the number of frequent itemsets generated is as follows-

Dataset	Total frequent itemsets
Baseline	62
Variant 1	4991
Variant 2	4525
Variant 3	10700
Variant 4	1350

Varying the number of cores-

The taskset command line utility has been used to limit the number of cores the process can use to run with. Accordingly, during each run, changes were made in the code to only allow those many as available cores.

```
taskset --cpu-list <cpu_range> ./driver
```

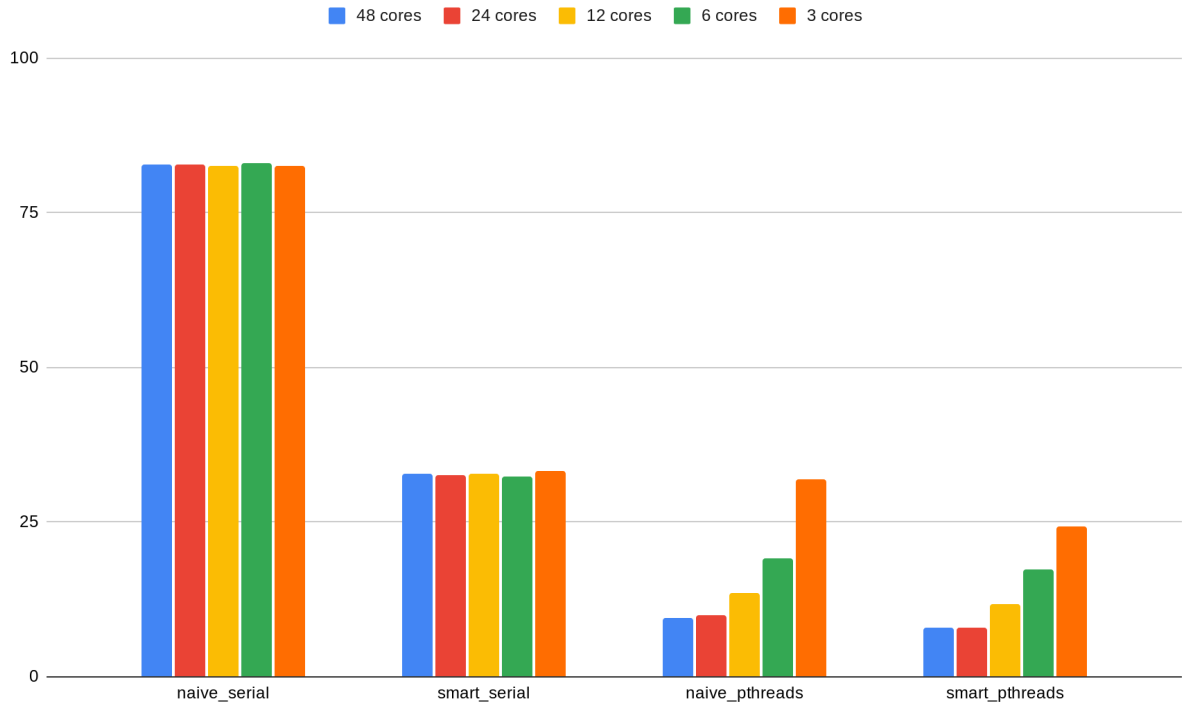
Example: `taskset --cpu-list 0-23 ./driver`

The parameters set were-

NUM_TX = 500
NUM_ITEMS = 50
PROBABILITY = 0.5
THRESHOLD = 20

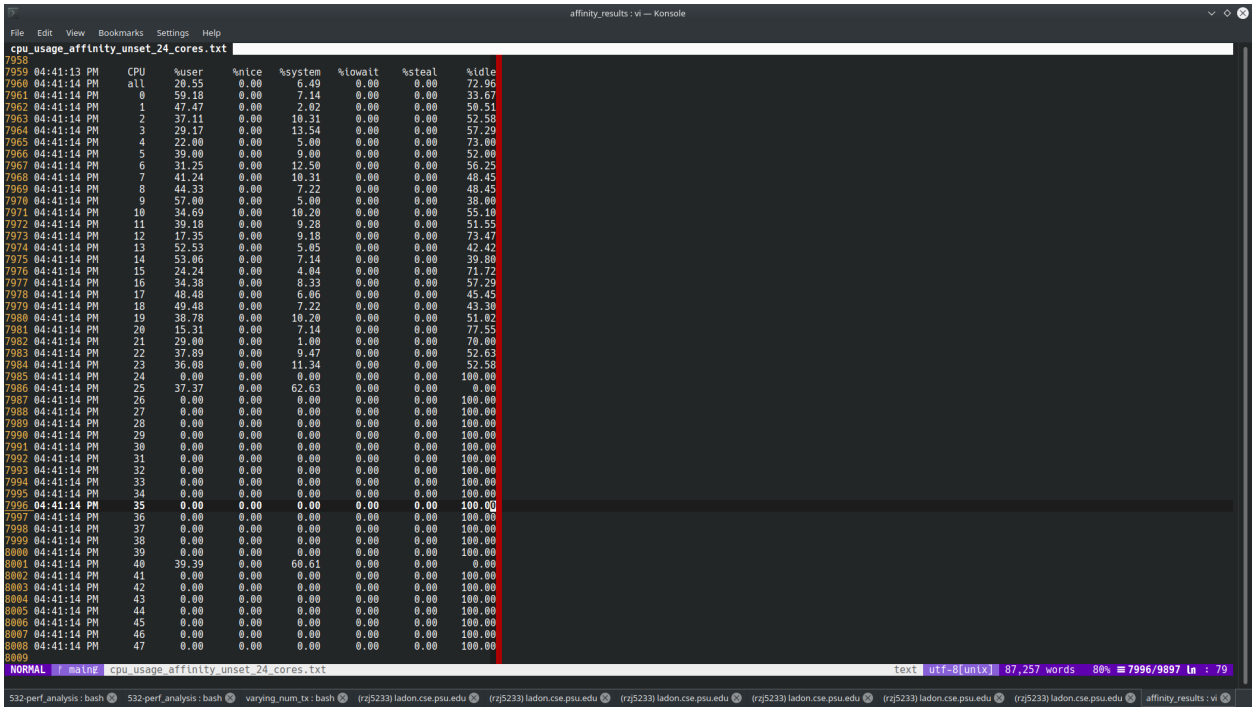
Number of cores and which ones specifically were used is as shown below-

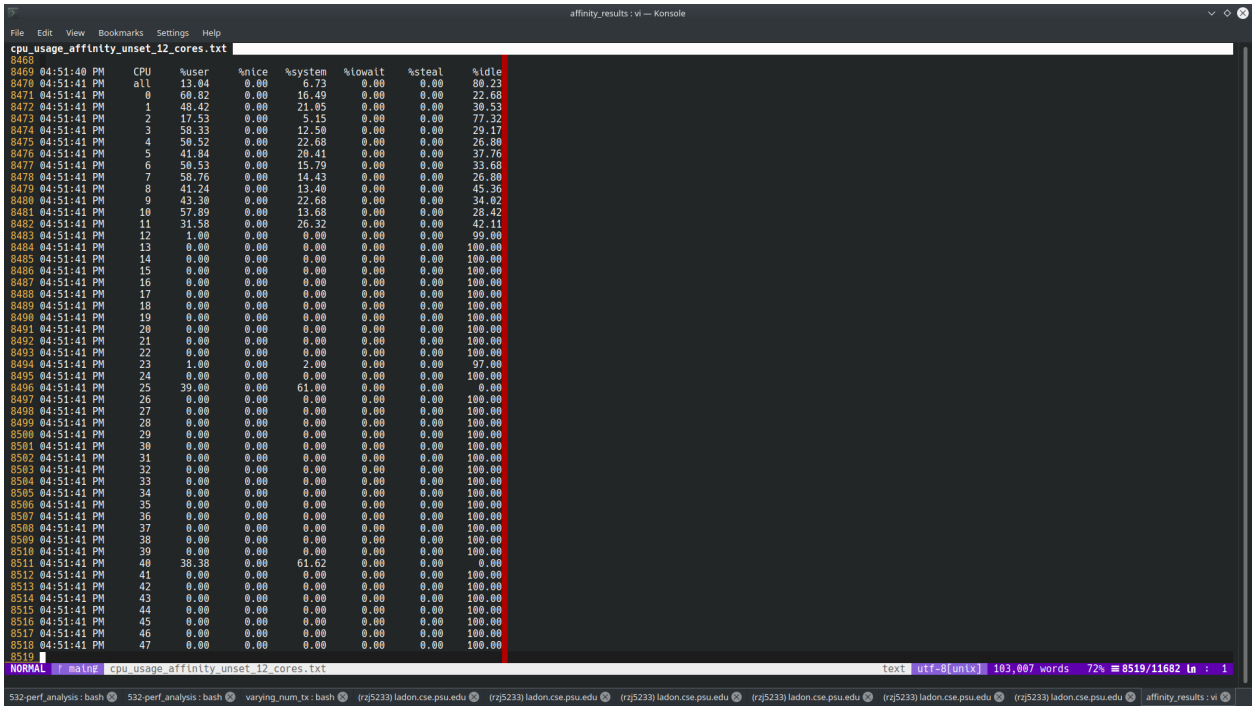
Number of cores	Core range
48	0 - 47
24	0 - 23
12	0 - 11
6	0 - 5
3	0 - 2

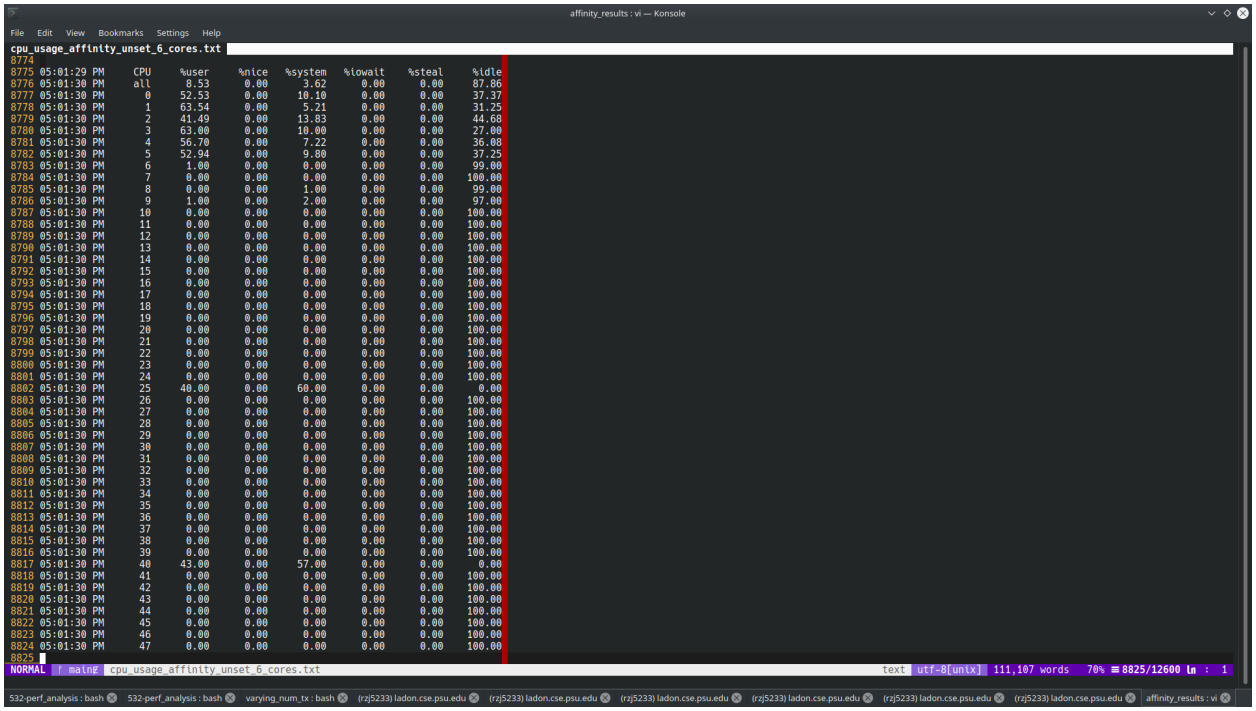


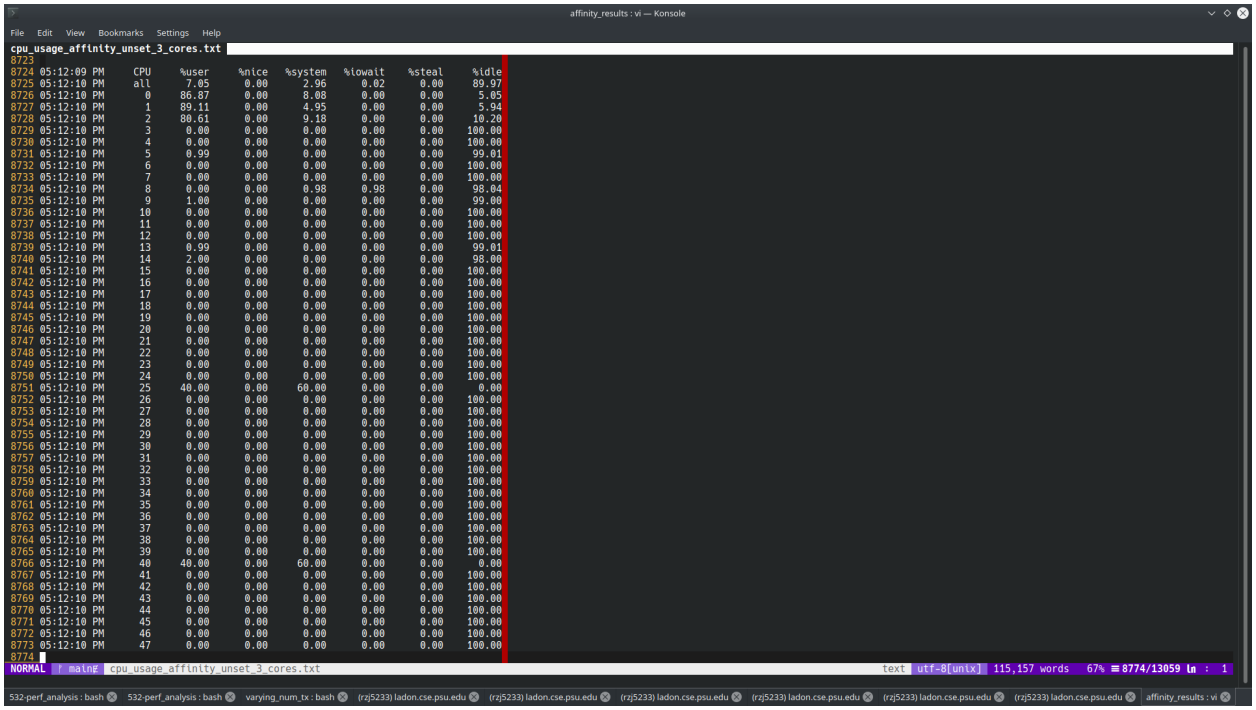
As seen in the graph above, the sequential implementation does not get affected by reducing the number of allowed cores since it anyways runs on just a single core. On the other hand, in the case of the pthreads implementation, when the number of allowed cores is decreased, there is a linear increase in the execution time. This behaviour is as expected since the number of cores available for compute are less and hence it'll take more time to execute the same workload. The smart implementation shows lower execution times than the naive implementation in the obtained results, which is as expected. Finally, as can be observed, the pthreads implementation performs better than the serial one.

Restricted CPU usage based upon sar output as set by tasklet-




affinity_results : vi — Konsole
File Edit View Bookmarks Settings Help
cpu_usage_affinity_unset_12_cores.txt
8468
8469 04:51:40 PM CPU %user %nice %system %iowait %steal %idle
8470 04:51:41 PM all 13.04 0.00 6.73 0.00 0.00 80.23
8471 04:51:41 PM 0 60.82 0.00 16.49 0.00 0.00 22.68
8472 04:51:41 PM 1 48.42 0.00 21.05 0.00 0.00 30.53
8473 04:51:41 PM 2 17.53 0.00 5.15 0.00 0.00 77.32
8474 04:51:41 PM 3 58.33 0.00 12.50 0.00 0.00 29.17
8475 04:51:41 PM 4 50.52 0.00 22.68 0.00 0.00 26.80
8476 04:51:41 PM 5 41.84 0.00 20.41 0.00 0.00 37.76
8477 04:51:41 PM 6 50.53 0.00 15.79 0.00 0.00 33.68
8478 04:51:41 PM 7 58.76 0.00 14.43 0.00 0.00 26.80
8479 04:51:41 PM 8 41.24 0.00 13.40 0.00 0.00 45.36
8480 04:51:41 PM 9 43.30 0.00 22.68 0.00 0.00 34.02
8481 04:51:41 PM 10 57.89 0.00 13.68 0.00 0.00 28.42
8482 04:51:41 PM 11 31.58 0.00 26.32 0.00 0.00 42.11
8483 04:51:41 PM 12 1.00 0.00 0.00 0.00 0.00 99.00
8484 04:51:41 PM 13 0.00 0.00 0.00 0.00 0.00 100.00
8485 04:51:41 PM 14 0.00 0.00 0.00 0.00 0.00 100.00
8486 04:51:41 PM 15 0.00 0.00 0.00 0.00 0.00 100.00
8487 04:51:41 PM 16 0.00 0.00 0.00 0.00 0.00 100.00
8488 04:51:41 PM 17 0.00 0.00 0.00 0.00 0.00 100.00
8489 04:51:41 PM 18 0.00 0.00 0.00 0.00 0.00 100.00
8490 04:51:41 PM 19 0.00 0.00 0.00 0.00 0.00 100.00
8491 04:51:41 PM 20 0.00 0.00 0.00 0.00 0.00 100.00
8492 04:51:41 PM 21 0.00 0.00 0.00 0.00 0.00 100.00
8493 04:51:41 PM 22 0.00 0.00 0.00 0.00 0.00 100.00
8494 04:51:41 PM 23 1.00 0.00 2.00 0.00 0.00 97.00
8495 04:51:41 PM 24 0.00 0.00 0.00 0.00 0.00 100.00
8496 04:51:41 PM 25 39.00 0.00 61.00 0.00 0.00 0.00
8497 04:51:41 PM 26 0.00 0.00 0.00 0.00 0.00 100.00
8498 04:51:41 PM 27 0.00 0.00 0.00 0.00 0.00 100.00
8499 04:51:41 PM 28 0.00 0.00 0.00 0.00 0.00 100.00
8500 04:51:41 PM 29 0.00 0.00 0.00 0.00 0.00 100.00
8501 04:51:41 PM 30 0.00 0.00 0.00 0.00 0.00 100.00
8502 04:51:41 PM 31 0.00 0.00 0.00 0.00 0.00 100.00
8503 04:51:41 PM 32 0.00 0.00 0.00 0.00 0.00 100.00
8504 04:51:41 PM 33 0.00 0.00 0.00 0.00 0.00 100.00
8505 04:51:41 PM 34 0.00 0.00 0.00 0.00 0.00 100.00
8506 04:51:41 PM 35 0.00 0.00 0.00 0.00 0.00 100.00
8507 04:51:41 PM 36 0.00 0.00 0.00 0.00 0.00 100.00
8508 04:51:41 PM 37 0.00 0.00 0.00 0.00 0.00 100.00
8509 04:51:41 PM 38 0.00 0.00 0.00 0.00 0.00 100.00
8510 04:51:41 PM 39 0.00 0.00 0.00 0.00 0.00 100.00
8511 04:51:41 PM 40 38.38 0.00 61.62 0.00 0.00 0.00
8512 04:51:41 PM 41 0.00 0.00 0.00 0.00 0.00 100.00
8513 04:51:41 PM 42 0.00 0.00 0.00 0.00 0.00 100.00
8514 04:51:41 PM 43 0.00 0.00 0.00 0.00 0.00 100.00
8515 04:51:41 PM 44 0.00 0.00 0.00 0.00 0.00 100.00
8516 04:51:41 PM 45 0.00 0.00 0.00 0.00 0.00 100.00
8517 04:51:41 PM 46 0.00 0.00 0.00 0.00 0.00 100.00
8518 04:51:41 PM 47 0.00 0.00 0.00 0.00 0.00 100.00
8519
NORMAL f main@ cpu_usage_affinity_unset_12_cores.txt text utf-8[unix] 103,007 words 72% ≡ 8519/11682 ln : 1
532-perf_analysis : bash 532-perf_analysis : bash varying_num_tx : bash (rzj5233) ladon.cse.psu.edu (rzj5233) ladon.cse.psu.edu (rzj5233) ladon.cse.psu.edu (rzj5233) ladon.cse.psu.edu (rzj5233) ladon.cse.psu.edu (rzj5233) ladon.cse.psu.edu affinity_results : vi


affinity_results : vi — Konsole
File Edit View Bookmarks Settings Help
cpu_usage_affinity_unset_6_cores.txt
8774
8775 05:01:29 PM CPU %user %nice %system %iowait %steal %idle
8776 05:01:30 PM all 8.53 0.00 3.62 0.00 0.00 87.86
8777 05:01:30 PM 0 52.53 0.00 10.10 0.00 0.00 37.37
8778 05:01:30 PM 1 63.54 0.00 5.21 0.00 0.00 31.25
8779 05:01:30 PM 2 41.49 0.00 13.83 0.00 0.00 44.68
8780 05:01:30 PM 3 63.00 0.00 10.00 0.00 0.00 27.00
8781 05:01:30 PM 4 56.70 0.00 7.22 0.00 0.00 36.08
8782 05:01:30 PM 5 52.94 0.00 9.80 0.00 0.00 37.25
8783 05:01:30 PM 6 1.00 0.00 0.00 0.00 0.00 99.00
8784 05:01:30 PM 7 0.00 0.00 0.00 0.00 0.00 100.00
8785 05:01:30 PM 8 0.00 0.00 1.00 0.00 0.00 99.00
8786 05:01:30 PM 9 1.00 0.00 2.00 0.00 0.00 97.00
8787 05:01:30 PM 10 0.00 0.00 0.00 0.00 0.00 100.00
8788 05:01:30 PM 11 0.00 0.00 0.00 0.00 0.00 100.00
8789 05:01:30 PM 12 0.00 0.00 0.00 0.00 0.00 100.00
8790 05:01:30 PM 13 0.00 0.00 0.00 0.00 0.00 100.00
8791 05:01:30 PM 14 0.00 0.00 0.00 0.00 0.00 100.00
8792 05:01:30 PM 15 0.00 0.00 0.00 0.00 0.00 100.00
8793 05:01:30 PM 16 0.00 0.00 0.00 0.00 0.00 100.00
8794 05:01:30 PM 17 0.00 0.00 0.00 0.00 0.00 100.00
8795 05:01:30 PM 18 0.00 0.00 0.00 0.00 0.00 100.00
8796 05:01:30 PM 19 0.00 0.00 0.00 0.00 0.00 100.00
8797 05:01:30 PM 20 0.00 0.00 0.00 0.00 0.00 100.00
8798 05:01:30 PM 21 0.00 0.00 0.00 0.00 0.00 100.00
8799 05:01:30 PM 22 0.00 0.00 0.00 0.00 0.00 100.00
8800 05:01:30 PM 23 0.00 0.00 0.00 0.00 0.00 100.00
8801 05:01:30 PM 24 0.00 0.00 0.00 0.00 0.00 100.00
8802 05:01:30 PM 25 40.00 0.00 60.00 0.00 0.00 0.00
8803 05:01:30 PM 26 0.00 0.00 0.00 0.00 0.00 100.00
8804 05:01:30 PM 27 0.00 0.00 0.00 0.00 0.00 100.00
8805 05:01:30 PM 28 0.00 0.00 0.00 0.00 0.00 100.00
8806 05:01:30 PM 29 0.00 0.00 0.00 0.00 0.00 100.00
8807 05:01:30 PM 30 0.00 0.00 0.00 0.00 0.00 100.00
8808 05:01:30 PM 31 0.00 0.00 0.00 0.00 0.00 100.00
8809 05:01:30 PM 32 0.00 0.00 0.00 0.00 0.00 100.00
8810 05:01:30 PM 33 0.00 0.00 0.00 0.00 0.00 100.00
8811 05:01:30 PM 34 0.00 0.00 0.00 0.00 0.00 100.00
8812 05:01:30 PM 35 0.00 0.00 0.00 0.00 0.00 100.00
8813 05:01:30 PM 36 0.00 0.00 0.00 0.00 0.00 100.00
8814 05:01:30 PM 37 0.00 0.00 0.00 0.00 0.00 100.00
8815 05:01:30 PM 38 0.00 0.00 0.00 0.00 0.00 100.00
8816 05:01:30 PM 39 0.00 0.00 0.00 0.00 0.00 100.00
8817 05:01:30 PM 40 43.00 0.00 57.00 0.00 0.00 0.00
8818 05:01:30 PM 41 0.00 0.00 0.00 0.00 0.00 100.00
8819 05:01:30 PM 42 0.00 0.00 0.00 0.00 0.00 100.00
8820 05:01:30 PM 43 0.00 0.00 0.00 0.00 0.00 100.00
8821 05:01:30 PM 44 0.00 0.00 0.00 0.00 0.00 100.00
8822 05:01:30 PM 45 0.00 0.00 0.00 0.00 0.00 100.00
8823 05:01:30 PM 46 0.00 0.00 0.00 0.00 0.00 100.00
8824 05:01:30 PM 47 0.00 0.00 0.00 0.00 0.00 100.00
8825
NORMAL f main@ cpu_usage_affinity_unset_6_cores.txt text utf-8[unix] 111,107 words 70% ≡ 8825/12600 ln : 1
532-perf_analysis : bash 532-perf_analysis : bash varying_num_tx : bash (rzj5233) ladon.cse.psu.edu (rzj5233) ladon.cse.psu.edu (rzj5233) ladon.cse.psu.edu (rzj5233) ladon.cse.psu.edu (rzj5233) ladon.cse.psu.edu (rzj5233) ladon.cse.psu.edu affinity_results : vi



As expected, in case of the sequential implementation, one of the cores among the restricted set is utilized (**note:** image not shown above).

5.8. Summarizing the linux utilities used

1. **sar-** can be used to monitor cpu usage characteristics (one advantage of this command is that its output can be redirected).
2. **htop / top-** similar to sar, but these commands provide a lot of other information like running processes, which users are running which processes, etc.
3. **taskset-** can be used to run any other command by limiting / restricting the allowed cores for that command.
4. **nvprof-** a profiling tool that enables collection and viewing of profiling data from the command line (a part of the CUDA toolkit).

6. Programming challenges

One of the biggest challenges for us was to figure out what data structure to use in the CUDA kernel. Since we used vectors in cpu implementations, we were first inclined to have a vector version in CUDA. However, we decided to go for a simple linear array as adopted in the general play-code/tutorial examples of vector addition.

Another challenge was to directly adopt the textbook's parallel algorithm in CUDA. We were not able to figure out how to do the computations of level 3 and later with just one cuda kernel launch, particularly, how would the GPU share intermediate results with the CPU for bookkeeping when required, and storing all the levels in a common data structure shared by all GPU threads. So, we used another approach as described in Section 3.5.

7. Hardware bottlenecks

1. All the cores in CPU are generally not fully available as we found some other processes running for long duration (like days).
2. Our GPU implementation could benefit from concurrent kernel launches. However, it is difficult to program, and usually kernels are executed sequentially by the GPU.
3. Unlike pthreads where a CPU thread can launch new threads dynamically to scale up when the computation for a thread increases, it is tricky for a GPU thread to efficiently launch another kernel.
4. The multicore analysis as shown in Section 5.7 suggests that the pthread implementation doesn't get benefit when going from 24 cores to 48 cores.

8. Limitations and future directions

1. More intelligent use of cpu affinity needs to be pursued. Currently, only a simple round robin runtime allocation mechanism has been used, which might not always be optimal since the results obtained do not show significant improvement with cpu affinity set, which ideally should be the case.
2. In the current pthreads implementation, there is parallelism introduced only at level 2's equivalence classes. But within each thread that handles this equivalence class, at further levels, all the computation happens in this single thread sequentially, though asynchronously. A further scope for optimization via more parallelism opportunities would be to launch new child threads for equivalence classes generated at each level, leading to a tree of threads.
3. An interesting way to find the least frequent items could be by inverting the matrix (meaning: setting all 1 to 0 and all 0 to 1). Sales experts could use this analysis to find certain items to remove from the store which could help in making space for new or popular items.
4. With a larger database size, and dense matrix, the GPU should show better speedups compared to a multi-core CPU. This can be determined as part of future study.
5. Writing efficient CUDA code - there are a lot of functions which CUDA programming supports to efficiently use GPUs. Our implementation was just a beginning endeavor to demonstrate the benefit of GPUs. However, we carefully understand this, and we aim to write efficient code which industry can adopt as part of their products.
6. Since the application heavily relies on reading and writing large arrays, there is a lot of scope on doing memory optimizations. Particularly, one can use the benefits of data locality, prefetching, and reducing memory divergence. Learning to profile parts of the program to identify the bottlenecks (API calls or long execution time code blocks) and correctly tuning the application should be a wise way to improve our application performance. We plan to follow this as part of future work.
7. Heterogenous style computing could be adopted based on the work to be done. If the work has less parallelization opportunity, it could be a good idea to run it on a CPU (which generally has limited cores). On the other hand, if the work has more

parallelization opportunities, it could be beneficial to prioritize it to run on the GPU. For eg: if the equivalence class has less than 20 items sets, run it on CPU, else run it on the GPU.

9. References

[1] ExampleApps CSE 532 Course Lecture Slides PPT

[2] Parallel Computer Architecture: A Hardware/Software Approach, Pgs 80 - 81, 179 - 181