Speedy-Panda: fast evaluation of the PaNDa$^+$ algorithm

Giulio Zausa
Ca' Foscari University of Venice
870040@stud.unive.it

For this project we are going to implement the PaNDa$^+$ [1] algorithm for Approximate Top–K Frequent Binary Patterns Mining in Noisy Datasets, exploiting parallelization techniques to achieve a processing time speedup. Our goal is to be able to process large datasets faster, while preserving the correctness of the results. In this document we will describe our approach, the implementation details, the results and the verification strategy.

I. Introduction

A. Problem Description

The problem that we are trying to solve is the extraction of approximate top-k patterns that are able to concisely describe a noisy transactional binary input data. The input data is made by N transactions and M items, and it can be represented by a *binary* matrix $\mathcal{D} \in \{0,1\}^{N \times M}$ where $\mathcal{D}(i,j) = 1$ if the i–th item occurs in the j–th transaction, and $\mathcal{D}(i,j) = 0$ otherwise. An *approximate pattern* P that we want to extract is denoted by a pair of binary vectors $P = \langle P_I, P_T \rangle$, where $P_I \in \{0,1\}^M$ and $P_T \in \{0,1\}^N$. Our goal is to extract a set of patterns $\overline{\Pi} = \{P_1, \ldots, P_{|\Pi|}\}$ that covers the dataset, minimising a generalized cost function J that takes into account noise and pattern complexity.

$$\overline{\Pi}_k = \underset{\Pi_k}{\operatorname{argmin}}\, J(\Pi_k, \mathcal{D}) \tag{1}$$

$$\mathcal{N} = \bigvee_{P \in \Pi} (P_T \cdot T_I^{\mathrm{T}}) \ \veebar \ \mathcal{D} \tag{2}$$

$$J = (\Pi_k, \mathcal{D}, \rho, \gamma_P, \gamma_{\mathcal{N}}) = \rho \cdot \sum_{P \in \Pi_k} \gamma_P(P) \ + \ \gamma_{\mathcal{N}}(\mathcal{N}) \tag{3}$$

For our implementation we decided to use a specific fixed cost function $J_P^{\overline{\rho}}$ to have more control over performance optimizations, such as inlining and vectorization. Our cost function is a variation of the one proposed in [2] and [3], taking into account noise and pattern set complexity with a weight factor [1].

$$J_P^{\overline{\rho}} = (\Pi_k, \mathcal{D}) = \overline{\rho} \cdot \sum_{P \in \Pi_k} (\|P_T\| + \|P_I\|) \ + \ \|\mathcal{N}\| \tag{4}$$

Since [4] proved that the problem solution is NP–hard, we are using the PaNDa$^+$ algorithm to approximate its solution, using a greedy approach. A general description of the algorithm can be found in [1], we will omit it to describe only our implementation details and performance evaluation.

B. Implementation method

To build our highly–performant implementation of the algorithm we decided to use a standard approach, using the C++ language and OpenMP to manage parallelism. This allowed us to tune the performance at a lower level than other languages like Python or Java, still having the benefits of high level utilities such as STL. To achieve better run time performance we exploited parallel data processing, both using instruction–level parallelism (*i.e. SIMD*) and multithreading. We decided to use the OpenMP [5] library to perform the multithreading parallelisation, since

it allowed us to use parallel processing techniques without changing the serial code too much.

Our implementation can read transactional datasets from *ASCII* text files, where every line represents a transaction and every item is an integer separated by a single space. Every integer number of the dataset represent an item in the binary matrix i.e. $\mathcal{D}(i, j) = 1$ where i is the row (the line number in the file) and j is the integer number written in *ASCII* format. Our implementation takes the file path as command line parameter together with other optional parameters, such the maximum number of patterns k and the complexity weight $\overline{\rho}$. Other info about the command line parameters can be found in the readme and by using the `-h` option.

The code for our implementation can be found on our GitHub repository[1].

C. Datasets and Verification

While developing the application we applied Test Driven Development *(TDD)* principles to ensure correctness of our results. We used the *doctest*[2] library to perform automated unit tests for every single tasks performed by the algorithm. This allowed us to ensure the correctnes of the algorithm, especially to find out race conditions that arised during the parallel implementation.

To check the quality of the patterns we also compared our results with the ones given by the implementation of PaNDa provided in [3], using its running times as baseline. To ensure a realistic performance evaluation we used the datasets provided in *Frequent Itemset Mining Dataset Repository*[3], such as `accidents.dat`[6] and `retail.dat`[7].

D. Performance measurement methods

To measure the performance metrics of our application we used several tools, such as Unix `time` utility, Apple's developer tool *Instruments* and we also instrumented the code with C++ `std::chrono` calls to measure the run times of specific code sections.

When performing optimizations we took into consideration the following metrics:

- **Total Run Time**: total running time and CPU time spent by the program processing a dataset, measured using Unix `time` and `std::chrono` instrumentations;
- **Functions CPU Time**: time spent in each function of the program using *Instruments*, to understand the most expensive parts of the algorithm to optimize, according to the Amdahl's law;
- **CPU Cache Misses**: number of CPU instructions skipped because of L3 cache misses using *Instruments*, reading from the `MEM_LOAD_RETIRED.L3_MISS` internal Intel CPU counter, as explained in [8]. We have chosen to consider only L3 cache misses since L1 and L2 misses are fast enough to be negligible under our workloads, and L3 misses causes RAM read, which is slow.

II. Algorithm Evolution

We will now describe the phases of our algorithm evolution, reasoning about how we made some design choices to gain better performances.

A. Naive Implementation

Before any optimization attempt we implemented the algorithm as it's described in [1]. This implementation can be found in the `naive` branch on the Git repository:

- The **dataset** is saved in memory as `std::vector<std::set<int>>`: every row of the vector is a transaction, and the items are saved in a int set. In practice this means that the vector is a linear array of pointers, and every set typically use a tree-like structure, ensuring $O(\log(n))$ access.
- We store the **patterns** in memory as `std::vector<Pattern>`, where `Pattern` is a struct

[1] https://github.com/giulioz/speedy-panda
[2] https://github.com/onqtam/doctest
[3] http://fimi.uantwerpen.be/data/

containing two `std::set<int>` both for items and transactions.

- As described in the paper, the function J to calculate the cost of a pattern set $\overline{\Pi}_k$ is called every time we want to check if we have a better pattern candidate to add, both in `findCore`, `endCore` and `panda`. Since the cost function loops over the entire dataset to find false positives and false negatives, its complexity is $O(kNM)$ (assuming $O(1)$) access in the pattern list and dataset, even though we know it's at least $O(\log(n))$ by STL implementation).

The naive implementation resulted in a very slow processing time, mostly because of the useless recalculation of the cost: it can be avoided keeping track of the false positives and false negatives in a smart way. To prove this reasoning we used the *Instruments* profiler: $\approx 75\%$ of the processing time were spent in the `costFunction()` function.

For this reason we were able to test this implementation only with the first 700 line of the `retail.dat` dataset, since processing the entire dataset would have took too much time (see times in Table I).

B. Data Structures Optimization

As second step for optimization, we tried to keep our naive approach changing only the STL containers involved:

- Patterns transactions and items are now stored using an `std::unordered_set`, which uses an hash table instead of a tree structure. This dramatically reduced the L3 cache misses and run times. This can be explained with the fact that standard set enforces ordering, requiring additional writes and reorganization with every item and transaction insert, which is a frequent operation.
- We tried to store the transactions rows using `std::unordered_set`, `std::vector` and `std::vector` keeping ascending order to allow binary search. While the first increased the cache miss count (probably due to the much larger data involved), the ordered vector approach optimized the run time the most, since linear vectors are better for cache than trees.

The effect of these optimizations for run times and cache misses can be seen in Table I. Overall these design decisions allowed an average $\approx$ 1.5x speedup and reduced the L3 cache misses by $\approx 70\%$.

C. Inline Cost Calculation

As we have seen in section II-A, most of the processing time is spent recalculating the pattern set cost over and over, even though it can be calculated more efficently. The first optimization that we can introduce is caching the previous candidate pattern cost in `findCore`, `extendCore` and `panda`. The second optimization the we can introduce is avoiding looping over the entire dataset to calculate the noise: looking at equation 4 and 2 we can see how every time we add a transaction or item to a pattern we only need to keep track of the difference in false positives and false negatives, which is fairly easy considering these rules:

- Adding a new transaction i in a pattern $\overline{\Pi}$ subtracts $\sum_M^j \mathcal{D}_{\mathcal{R}}(i,j)$ from false negatives, i.e. every item that is now covered by the added transaction;
- Adding a new item j in a pattern $\overline{\Pi}$ subtracts $\sum_N^i \mathcal{D}_{\mathcal{R}}(i,j)$ from false negatives, i.e. every item in all transactions that is now covered by the added item;
- Adding a new transaction i in a pattern $\overline{\Pi}$ adds $\sum_M^j 1 - \mathcal{D}_{\mathcal{R}}(i,j)$ to false positives, i.e. every item that is NOT present in the added transaction;
- Adding a new item j in a pattern $\overline{\Pi}$ adds $\sum_N^i 1 - \mathcal{D}_{\mathcal{R}}(i,j)$ to false positives, i.e. every item in all transactions that NOT covered by the added item;

Keeping the false negatives and false positives count during computation, calculating only the differences between every pattern, increases the computation speed by

Implementation	findCore (ms)	extendCore (ms)	Total CPU Time (ms)	L3 Misses
Baseline [3]	N/A	N/A	30	3505
std::set (patt.)	16291	20490	36634	896962
std::unordered_set	13440	16805	30110	355594
std::unordered_set (trans.)	12930	16127	28820	448013
std::vector + std::find	12765	16278	28730	322782
std::vector + std::binary_search	11267	14308	25460	275801
Incremental Noise Count	364	44	415	4814
No Vector Erase	29	50	90	3769

Table I: Effects of different data structures using the naive algorithm, measured using the reduced `retail.dat` dataset to 700 transactions, under conditions described in section III-A, first row set for the patterns and second for the dataset. The last two rows shows the incremental noise count algorithm (section II-C).

a factor of $\approx$ 60x, as seen in Table I. We can do the same optimization even in the `notTooNoisy` function.

Another optimization that we can apply is avoid calling the `erase` method on STL's vectors, since it's an expensive operation when is repeated multiple times in a loop. Instead, we create a new vector inserting all the element not to be removed. We applied this optimization in `findCore` and `removePattern`, achieving another $\approx$ 3.2x speedup, for a total $\approx$ 238x speedup compared to the original naive version.

We then tested the optimized algorithm with the full `retail.dat` dataset and we obtained the results shown in Table II.

D. Multi–Thread Parallelism

After optimizing as much as possible the single core performances, we tried to exploit multi–core parallel computation to have better performances. To do so first of all we analyzed the algorithm behaviour, to find tasks that could go in parallel. Unfortunately, we quickly found a lot of data dependencies:

- It's not possible to partition the input dataset and then merge the results later;
- `extendCore` cannot run before `findCore` results and vice versa, otherwise it would produce different results;
- Even inside `extendCore` and `findCore` it's not possible to advance to another candidate before knowing the current best cost, since the algorithm is greedy. It would be possible to calculate all the possibilities and select them later in mutual exclusion, but this would only increase the total CPU time.

All that was left to parallelize were the inner loops inside `extendCore` and `findCore`. We tried to apply OpenMP directives to the following loops (lines are referred to [1]):

1) **findCore** (line 6): Can run in parallel row–wise for group of transactions, every thread can accumulate a part of C_T and final merge can happen at the end with a critical section;
2) **findCore** (line 10): Same as line 6, since we changed the erase behaviour to create a new vector, that will be merged at the end in C_T^* using a critical section;
3) **extendCore** (line 5): Can run in parallel row–wise since to add a new transaction to a pattern it's needed to loop over all its items in the dataset to calculate the resulting noise. We used am OpenMP reduction clause to accumulate the partials without using an explicit

Implementation	findCore (s)	extendCore (s)	Total Time (s)	CPU Time (s)	L3 Misses
Baseline [3]	N/A	N/A	69	69	248,939,386
Incremental Count	152	501	649	649	1,106,247,033
No Vector Remove	47	155	201	201	447,400,682
Parallel (1,2,3,4)	54	87	142	1210	N/A
Parallel (1,2,4)	50	69	120	1019	N/A

Table II: Effects of the optimizations described in section II-C, measured using the full `retail.dat` dataset, under conditions described in section III-A

mutex;

4) **extendCore** (line 12): Can run in parallel column–wise since to add a new item to a pattern it's needed to loop over all its transactions in the dataset to calculate the resulting noise;
5) **notTooNoisyTransaction**: The first clause (every item j of P must be included in at least $(1-c) \cdot \|PT\|$ transactions of P) can be run in parallel item-wise or transaction-wise, using an OpenMP reduction clause for the `ok` flag or the `columnSum`.

We evalutated the performance gains parallelizing those loops, finding out that option n.3 only slows down the processing, since the number of items M is usually not large enough and using threads for that creates overhead. We measured the performance of the parallel version with 12 threads in Table II, measuring a total $\approx$ 1.7x speedup than the serial version.

For the `notTooNoisy` function we found out that the item-wise approach yields better performance, with a total speedup of $\approx$ 5.7x with 8 cores, even though this would scale only as long as there are more items than threads.

Since the `notTooNoisy` function slowed down the calculation a lot, from now our results are calculated with the noise threshold both set to 1.0, causing the function to perform an early exit, since with this threshold it will always return true.

III. Performance Evaluation

A. Measurement Setup

The test were done using an high–performance laptop, assuming performances like a medium–spec HPC server:

- Apple clang version 12.0.0 (clang-1200.0.32.28), options `-Xpreprocessor -fopenmp -march=native -finline-functions -O3`
- MacBook Pro 16–Inch 2019, macOS 11.1
- 14 nm, 64-bit Intel Core i7 "Coffee Lake" (I7-9750H), 6 cores with Hyper Threading, 256k L2 cache, 12 MB L3 cache, 2.6GHz clock, 4.5GHz Turbo Boost
- 32GB DDR4 RAM (2666MHz)

B. Single–Thread Performance

Single–Thread benchmark times, under various datasets, are described in Table III.

Dataset	Total Time (s)	CPU Time (s)
retail_stripped	0.14	0.14
retail	250.17	247.95
accidents	138.19	136.90
chess	0.09	0.09

Table III: Final implementation times limited to a single thread, under different datasets

C. Multi–Thread Performance

Multi–Thread benchmark times, under the `accidents.dat` dataset, are described in Table IV.

We can see how the total processing time goes down until the number of threads reaches the hardware capability of the test machine (12, 6–cores with Hyper Threading). Once the number of threads is over 12 we can see a serious slowdown from the context–switch overhead and the OpenMP barriers, which are frequent in our code. Since the parallelized parts account for $\approx 60\%$ of the processing time, we expect to see a sublinear speedup increasing the number or threads (equation 5).

$$T(N_{threads}) = T_{fixed} + \frac{T_{parallel}}{N_{threads}} \tag{5}$$

N Threads	Total (s)	CPU (s)	Speedup
1	138.19	136.90	1.00x
2	96.69	186.45	1.43x
3	83.52	237.89	1.65x
4	76.99	286.44	1.79x
6	72.74	408.87	1.90x
8	64.19	477.05	2.15x
10	62.94	580.35	2.20x
12	63.93	680.62	2.16x
14	394.18	1087.31	0.35x
18	827.13	1012.05	0.17x
24	1184.39	1121.56	0.12x

Table IV: Final implementation times with different number of threads, under the `accidents.dat` dataset

We can compare the theoretical execution times from equation 5 with the real ones in Figure 2. We can see how the benchmarks closely follow the expected results by the theoretical model, until the number of threads grows over the machine capacity.

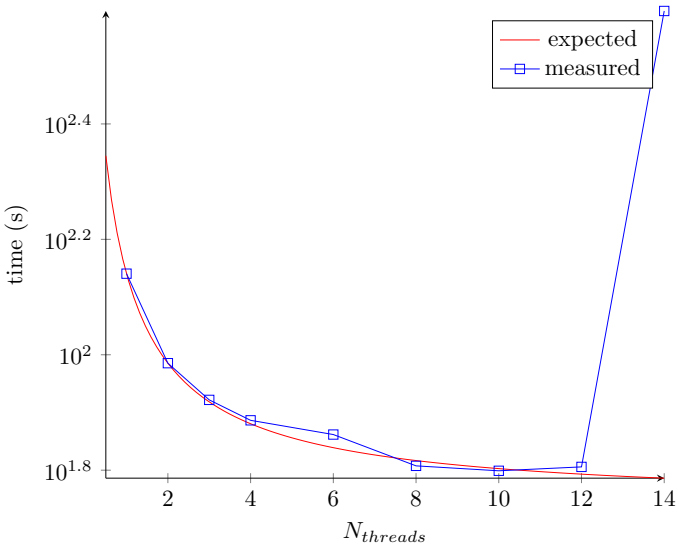

Figure 1: A log plot of the run times from Table IV

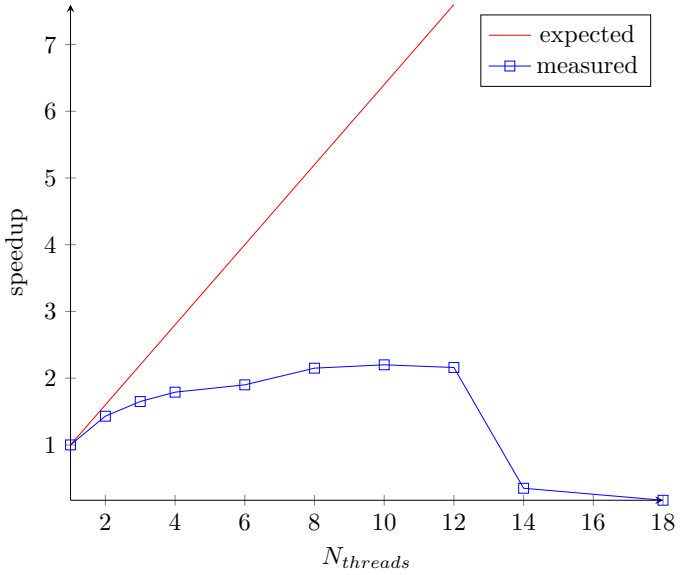

Figure 2: A plot of the speedup from Table IV

IV. Conclusion

We have discussed our implementation of the PaNDa$^+$ [1] algorithm, explaining our design choices, their effects on cache and processing time, and showing how it can process data in parallel using threads. We tested our implementation, showing how it can achieve an almost linear speedup increasing the number of threads, processing data $\approx$ 150x faster than a naive algorithm.

References

[1] C. Lucchese, S. Orlando, and R. Perego. "A Unifying Framework for Mining Approximate Top-k Binary Patterns". In: *IEEE Transactions on Knowledge and Data Engineering* 26 (2014), pp. 2900–2913.

[2] C. Lucchese, S. Orlando, and R. Perego. "A generative pattern model for mining binary datasets". In: *SAC '10*. 2010.

[3] C. Lucchese, S. Orlando, and R. Perego. "Mining Top-K Patterns from Binary Datasets in Presence of Noise". In: *SDM*. 2010.

[4] P. Miettinen et al. "The Discrete Basis Problem". In: *IEEE Transactions on Knowledge and Data Engineering* 20 (2008), pp. 1348–1362.

[5] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming". In: 1998.

[6] Karolien Geurts et al. "Profiling high frequency accident locations using associations rules". In: 2002.

[7] T. Brijs et al. "Using association rules for product assortment decisions: a case study". In: *KDD '99*. 1999.

[8] Hadi Brais. *An Introduction to the Cache Hit and Miss Performance Monitoring Events*. 2019. URL: https://hadibrais.wordpress.com/2019/03/26/an-introduction-to-the-cache-hit-and-miss-performance-monitoring-events/ (visited on 12/22/2020).