A PROJECT REPORT
ON

"PARALLELIZATION OF APRIORI ALGORITHM"

COURSE:
CS301(HIGH PERFORMANCE COMPUTING)

UNDER THE GUIDANCE OF
PROF. BHASKAR CHAUDHURY

AUTHORS:

OMKAR DAMLE ID 201401114
RUSHIKESH NALLA ID 201401106

Contents

Chapter 1

Context

1.1 Brief Description of the problem

The problem involves parallelization of the Apriori algorithm. In data mining, Apriori is a classic algorithm for learning association rules. Apriori is designed to operate on databases containing transactions(for example, collections of items bought by customers). The apriori algorithm generates frequent item-sets which have frequency greater than the specified support count. Using the frequent item-sets, association rules can be generated. We have limited the parallelization problem to generation of the frequent item-sets.

1.2 Complexity of the algorithm

If the number of transactions is N, the number of candidates is M and the time required to check if a candidate is present in a transactions is w, then the complexity of the algorithm is O(N*M*w). However the number of candidates M = $2\hat{d}$ where d = maximum size of itemset(number of unique items). In the apriori algorithm the number of candidates is reduced by pruning techniques. However the worst case complexity remains O(N*M*w). The rule used for pruning is : For a given frequent item-set, all subsets of the item-set must themselves be frequent item-sets.

1.3 Hardware Details

We ran this problem on a cluster of **16 cores**. The hardware details are :

Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 16
On-line CPU(s) list: 0-15
Thread(s) per core:1
Core(s) per socket: 8
Socket(s): 2
NUMA node(s): 2
Vendor ID: GenuineIntel
CPU family: 6
Model: 62
Stepping: 4
CPU MHz: 1200.000
BogoMIPS: 3999.43

Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 20480K
NUMA node0 CPU(s): 0-7
NUMA node1 CPU(s): 8-15

1.4 Real World Applications

Apriori Algorithm is the most popular and useful algorithm of Association Rule Mining of Data Mining. Association Rule in Data Mining plays a important role in the process of mining data for frequent itemsets. Frequent patterns are the patterns that occur frequently in the data. Patterns can include itemsets, sequences and subsequences. A frequent itemset refers to a set of items that often appear together in a transactional data set. Objective of taking Apriori is to find frequent itemsets and to uncover the hidden information. Some specific examples of apriori algorithm are mentioned as follows (references are mentioned below each application)

1.Apriori Algorithm for Market Basket Analysis

Market basket analysis is an important component of analytical system in retail organizations to determine the placement of goods, designing sales promotions for different segments of customers to improve customer satisfaction and hence the profit of the supermarket. These issues for a leading supermarket are addressed using frequent itemset mining. The frequent itemsets are mined from the market basket database using the efficient Apriori algorithm and then the association rules are obtained. Some variant of apriori known as K-Apriori is also used.

References
Market Basket Analysis for a Supermarket based on Frequent Itemset Mining by Loraine Charlet Annie M.C. and Ashok Kumar D.

2.Apriori Algorithm for Network Forensics Analysis

The massive network data must be captured and analyzed in network forensics, and the data is often related, the application of Apriori algorithm is proposed for network forensics analysis. After capturing and filtering network data package , and the Apriori algorithm is used to mine the association rules according to the evidence relevance to build and update signature database of offense, current user behavior is judged legal or not through pattern match results of user behavior and association rules which are stored in databases. The crime behaviors are saved in evidence database , which can be used as primitive evidence for network forensics. The application can help to resolve the real -time, efficient and adaptable problems in network forensics.

References
The Application of Apriori Algorithm for Network Forensics Analysis by Xiuyu Zhong.

3.Apriori Algorithm for Finding Locally Frequent Diseases

The data mining techniques can be applied on medical data which has abundant scope to improve Quality of Service in Healthcare industry. Medical data mining techniques analyze latent medi-

cal attributes and the relationships among them to bring about expert decisions in curing diseases. Apriori data mining technique can be used to know the frequently occurring diseases in the local databases.The modified apriori algorithm takes medical dataset and minimum support as inputs and generated locally frequent diseases from given medical data. Prior to applying the algorithm, the data set is pre-processed to associate numerals to discrete values for easy processing. The algorithm generates statistics pertaining to frequent itemsets that satisfy minimum support provided.

References

Finding Locally Frequent Diseases Using Modified Apriori Algorithm by Mohammed Abdul Khaleel, Sateesh Kumar Pradhan, G.N.Dash.

4.Apriori Algorithm for crimes concerning women

A WEKA tool is used for extracting results which includes attributes such as Age of boy, Age of Girl, Relation, Section and for mining the current dataset for association rules using the weka associators. As Relation Attribute tells us about the what a relation a victim has with a accused. Datasets are taken from UCI repository and a session court on crimes against women. Applying Apriori on real dataset against crimes on women extracts hidden information that what age group is responsible for this and to find where the real culprit is hiding.

References

Execution of APRIORI Algorithm of Data Mining Directed Towards Tumultuous Crimes Concerning Women by Divya Bansal and Lekha Bamchu.

Chapter 2

Serial Algorithm

2.1 Serial Pseudo Code

Pass 1

1. Generate the candidate itemsets in C1
2. Save the frequent itemsets in L1

Pass k

1. Generate the candidate itemsets in Ck from the frequent itemsets in Lk-1
    1. Join Lk-1 p with Lk-1 q, as follows:
    insert into Ck
    select p.item1, p.item2, . . . , p.itemk-1, q.itemk-1
    from Lk-1 p, Lk-1 q
    where p.item1 = q.item1, . . . p.itemk-2 = q.itemk-2, p.itemk-1 $<$ q.itemk-1
    2. Generate all (k-1) - subsets from the candidate itemsets in Ck
    3. Prune all candidate itemsets from Ck where some (k-1) - subset of the candidate itemset is not in the
2. Scan the transaction database to determine the support for each candidate itemset in Ck
3. Save the frequent itemsets in Lk

2.2 Algorithm

A **frequent itemset** is an itemset whose support is greater than some user-specified minimum support (denoted Lk, where k is the size of the itemset). A **candidate itemset** is a potentially frequent itemset (denoted Ck, where k is the size of the itemset)

Firstly we generate a candidate itemset of size 1 and store the number of the times the candidate appears in the transactions. Then we generate the frequent itemset of size 1 by removing all the candidates whose count is less than the given support count. We generate a candidate itemset of size k from frequency itemsets of size k-1. Consider any two frequency itemsets p and q of size k-1, we generate candidate of size k if all the items of p and q match except the last item. This is done for all possible pairs of the frequent itemsets of size k-1. For these generated candidates of size k, pruning is done. In pruning, a candidate is removed if at least one of its k-1 size subsets is not a frequent itemset of size k-1. We scan the transaction database to determine the support of each candidate subset in Ck and save all the frequency itemset in Lk.

Apriori is an iterative algorithm and at each step a frequent item-set of size k is generated. In subsequent iterations the value of k increases by 1. The algorithm continues till no frequent item-set with size k is found.

For our problem the input is a text file containing the transaction data. Each line of the input file contains the IDs of the items which are included in the transaction. The line ends with '-1'. The support count is passed as a runtime parameter. The output of the problem is frequent item-sets of all sizes which have a support count greater than the threshold.

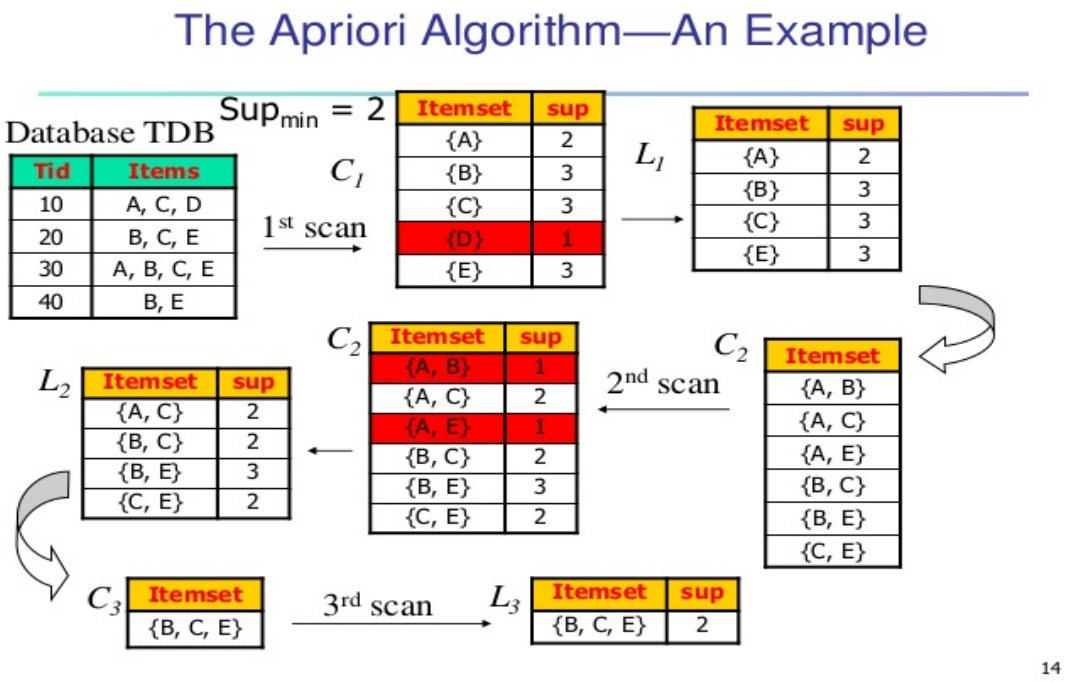


Figure 2.1: APRIORI EXAMPLE

Reference -
`http://www.slideshare.net/salahecom/data-mining-concepts-and-techniques-fp-basic`

In the above example input is the transaction database and the support count. Each scan k has item set of size k. In the first scan item D is removed as its count is less than support count and in the second scan (A,B) and (A,E) are removed. After 3 scans we get the frequent item set of size 3. The algorithm stops at this iteration because no more frequent itemsets can be produced.

2.3 Optimising the serial code

One important optimisation is in generate_C() function. While generating the possible candidates at kth iteration from the frequent itemsets of (k-1)st iteration, only a select combinations are considered. The selection criteria is that the frequent itemsets must differ in exactly one item and that should be the last item. This reduces the possible combinations substantially. This optimisation works because the selection criteria is a necessary condition for the candidate to escape pruning. So instead of removing some candidates in pruning, they are instead removed while generation only.

Consider Fig 2.2. The figure shows the effect of pruning the itemsets. All the supersets of an infrequent itemset are infrequent. Hence they need not be considered in future iterations.

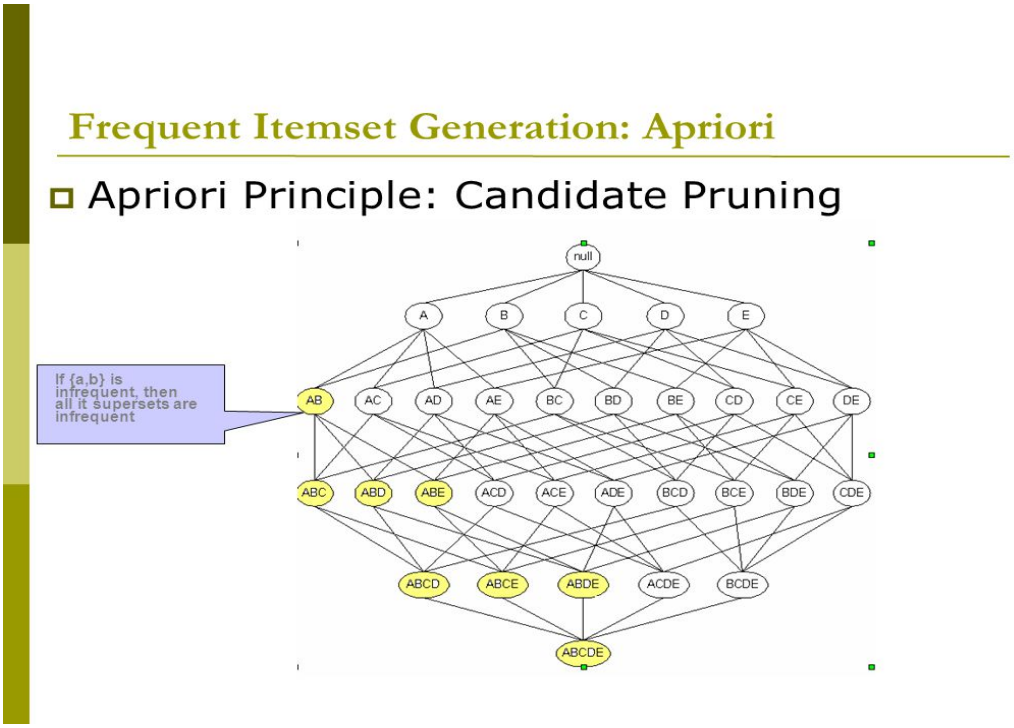


Figure 2.2: PRUNING IN APRIORI

Reference -

http://slideplayer.com/slide/5187921/

Chapter 3

Parallelizing the algorithm

3.1 Profiling information and Possible Speedup

We have used gprof for profiling and we have analysed the information for a couple of inputs.

1. Transactions - 5000, items - 20, support count - 1000
From the call graph information we saw that scan_d(part of the code which is parallel) took 86% of the time and from amdahl's law the maximum speedup that can be attained for 4 threads comes out to be 2.816. In fact this value is close to the result we obtained.

2. Transactions - 3000, items - 20, support count - 500
From the call graph information we saw that scan_d(part of the code which is parallel) took 82.2% of the time and from amdahl's law the maximum speedup that can be attained for 8 threads comes out to be 3.531. We achieved a speedup greater than 3.531.

3.2 Parallel pseudo code

Pass 1
1. Generate the candidate itemsets in C1
2. Save the frequent itemsets in L1

Pass k
1. Generate the candidate itemsets in Ck from the frequent itemsets in Lk-1
    1. Join Lk-1 p with Lk-1 q, as follows:
    insert into Ck
    select p.item1, p.item2, . . . , p.itemk-1, q.itemk-1
    from Lk-1 p, Lk-1 q
    where p.item1 = q.item1, . . . p.itemk-2 = q.itemk-2, p.itemk-1 $<$ q.itemk-1
    2. Generate all (k-1) - subsets from the candidate itemsets in Ck
    3. Prune all candidate itemsets from Ck where some (k-1) - subset of the candidate itemset is not in the frequent itemset Lk-1
2. **Distribute among p threads**
Scan the transaction database to determine the support for each candidate itemset in Ck
3. Save the frequent itemsets in Lk

3.3 Optimization strategy

3.3.1 Profiling

Our implementation of the apriori algorithm requires around 9 functions. Hence it is very important to recognise the part which needs to be parallelized. The Apriori algorithm is an iterative algorithm and hence there exists real loop dependency between the iterations. However each iteration itself takes a significant amount of time. (The number of iterations is between 5-15). Hence we decided to parallelize within each iteration. We used the gnu profiler-gprof tool. The call graph revealed important observations. We found that the database scanning function (scan_D()) takes up a major portion of the running time (around 90%). In fact, this function calls another function named set_count() which is responsible for the high amount of time spent in scan_D().

3.3.2 Parallelizing scan_D() function

So we concentrated our efforts on parallelizing the scan_D() function. This function iterates over all the transactions. For each transaction, it updates the counts of the candidates which are present in the transaction. Overall, a significant amount of time is spent in checking whether a candidate is present in the transaction or not. The transaction data was divided among the threads according to data decomposition principle. Each thread used a different file pointer for accessing the same database file. Each thread kept a local count of the candidate itemset. After each thread completed its work, it contained the counts of the candidates derived from the particular section of the database. Finally the counts for all the threads were added at the end in critical section. Note that the above process is for one particular iteration. It is not possible to introduce parallelization between iterations because there exists real loop dependency.

3.3.3 Other possibilies for parallelization

From the profiling analysis, the second most time consuming function was the generate_C(). It took around 5% of the total time. In the generate_C() function, possible candidates are generated from the frequent itemsets of previous iteration. This function involves a double loop which uses map iterators. Map iterators are bidirectional in nature. However Standard OpenMP doesn't bear with C++ iterators in general. OpenMP requires iterators to be random access iterators with constant time for random access. It also only permits $<$ and $<=$ or $>$ and $>=$ in test expressions of for loops, but not $!=$. As a result it was not possible to parallelize these loops with openMP. A possible solution is using Intel TBB (Thread building blocks). Reference site- https://www.threadingbuildingblocks.org/

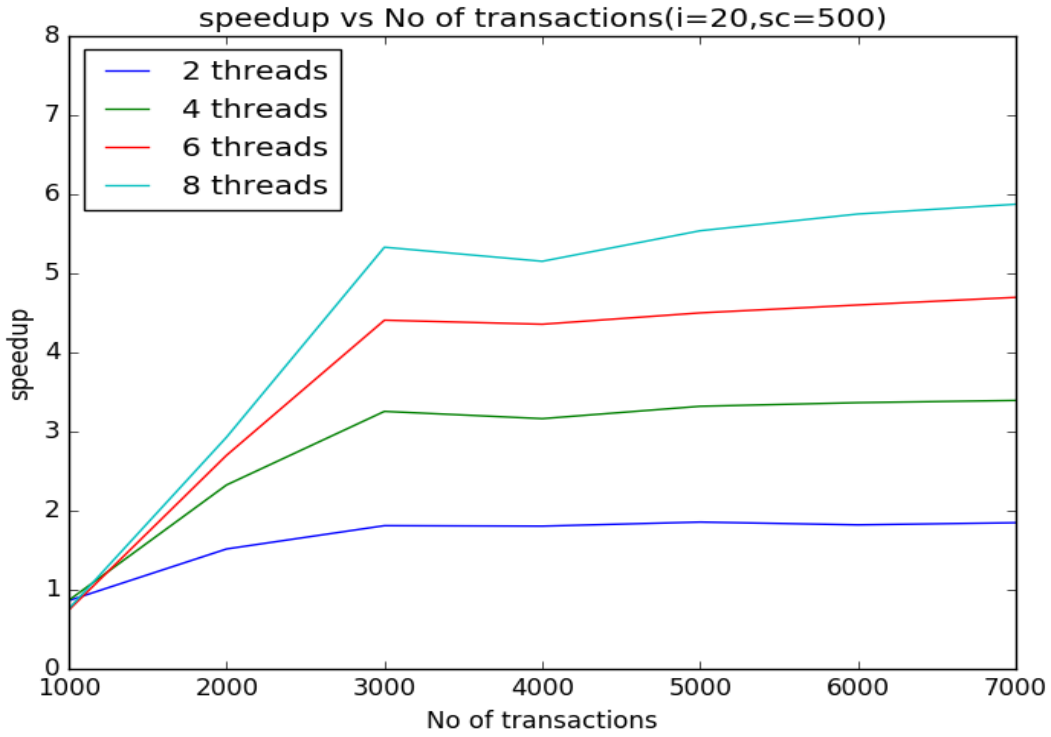
For the problem size which was feasible to run on maximum of 16 cores, other functions consumed negligible amount of time. Hence these functions if parallelised will not increase the speedup. However if the number of transactions is further increased then the profiling information should be taken to check if any other functions contribute to the time in a significant way.

Chapter 4

Results and observations

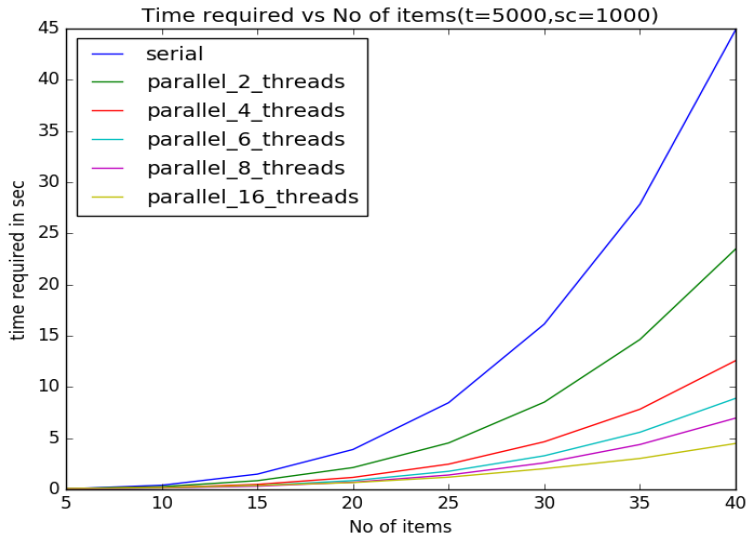
The input of the apriori algorithm is the transaction data. The transaction data was generated by a script which used random function of C language. The transaction data determines the problem size. The main parameters to be considered are number of transactions and number of individual items. In order to get an idea about the variation of speedup with problem size, we have considered both the parameters. The speedup graphs were plotted using matplotlib python library. The readings were taken using shell scripts.

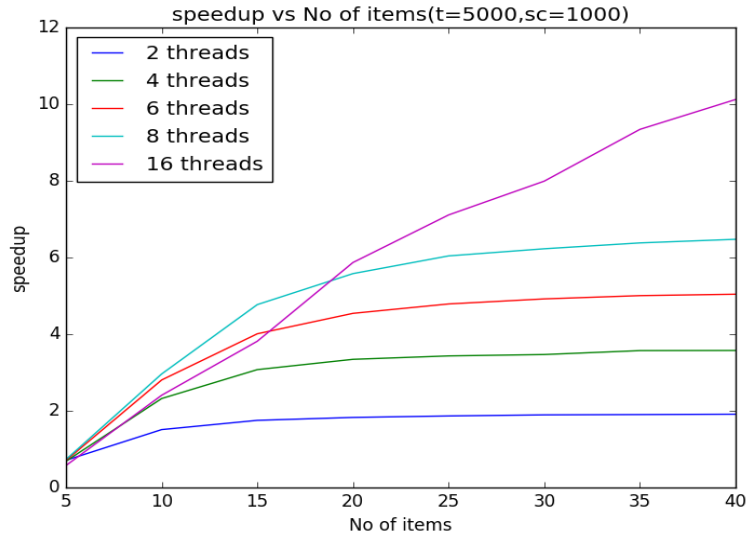
4.1 Speedup vs No of transactions



We can see that as the number of threads increases, speedup increases. Increasing the number of transactions is nothing but increasing the problem size and the speedup increases initially until it reaches its maximum and then after 3000 transactions it almost remains same. The main reason is that the time spent in serial and parallel sections increases proportionately as the transactions increase.

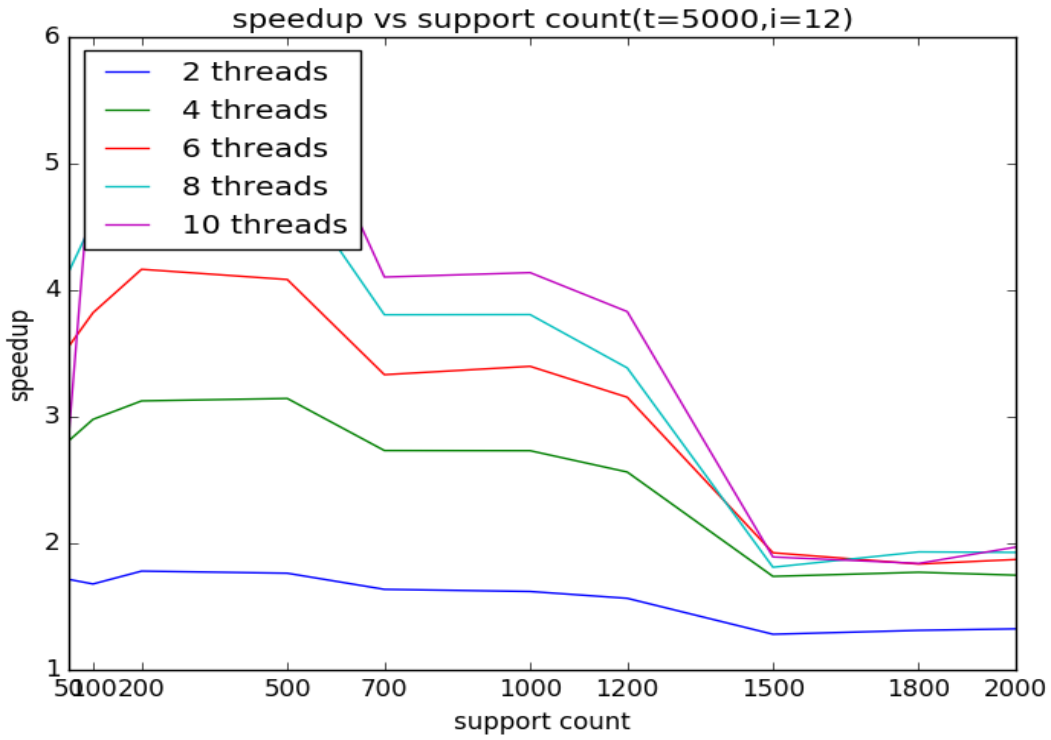
4.2 Speedup vs No of items





We can see that as the number of items increases, speedup increases. From the graph it is evident that upto 20 items speedup increases but later it stabilizes. By doing the gprof analysis we can see that scan_d takes 86% of the time and by amdahl's law the maximum speedup that can be attained for 4 threads comes out to be 2.816 and from the graph we can see that it has attained that speedup for 20 items. So even on increasing the items, there is not much change in the speedup.

4.3 Speedup vs Support Count



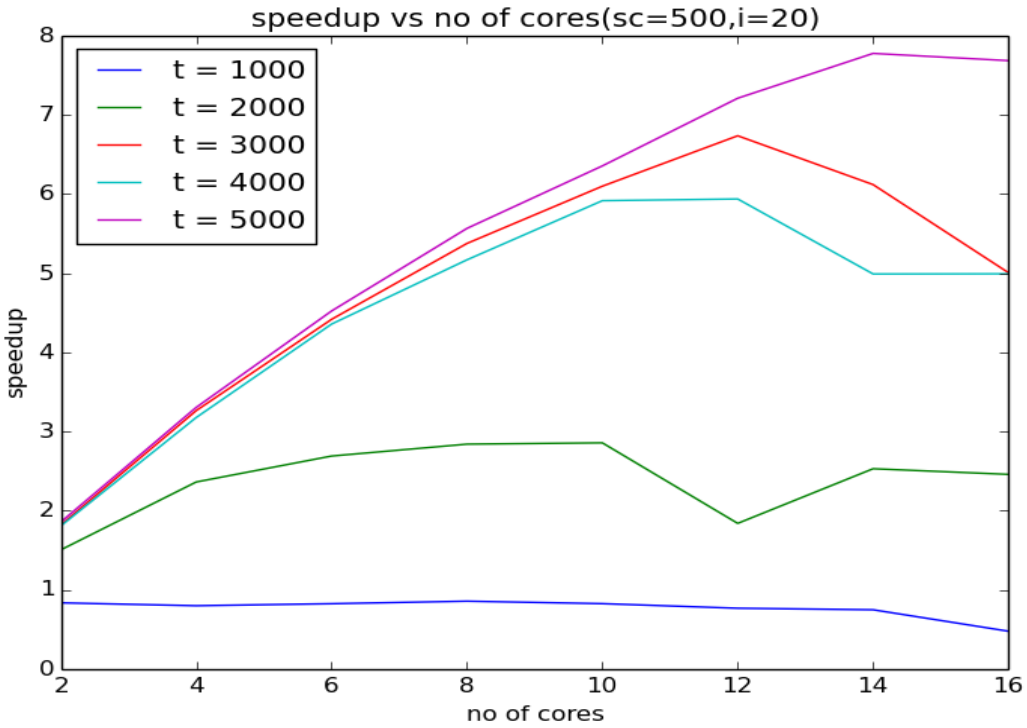
We can see that as support count increases, speedup decreases. This is quite intuitive because the number of iterations required to find the frequent item list decreases and the number of scans through the database also decreases (the part of the code which is parallelized) and hence the speedup decreases. We can also see that it is true for all the threads, the decreasing pattern follows. In various applications for example in grocery store the choice of support count is important because more the support count, in that many transactions a particular item set has occurred. When you calculate the confidence of an item, i.e when you buy item X and the possibility of buying item Y along with it, support count ensures the accuracy of the result.

#threads	2	4	6	8	10
Karp flatt metric	0.0751	0.0794	0.0756	0.0780	0.0768

Table 4.1: Karp flatt metric

4.4 Speedup vs number of cores

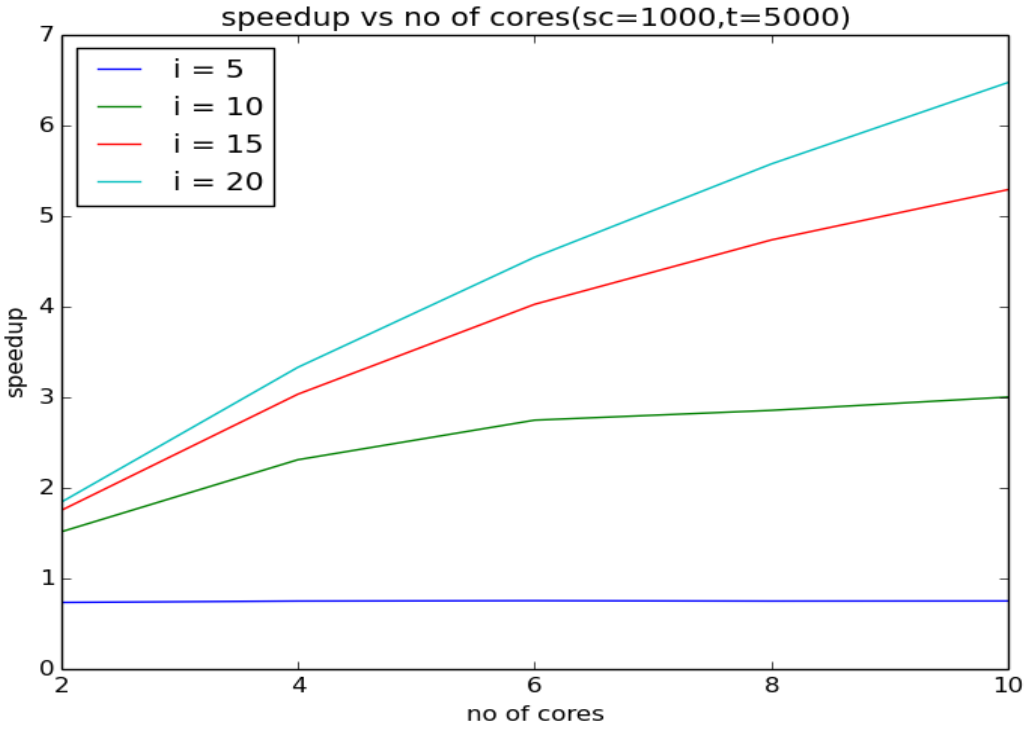
4.4.1 Varying transactions



As the number of cores increases, the speedup increases for larger input transactions. The problem size is directly proportional to the number of transactions. We can observe that for lower problem sizes, the speedup saturates much faster. In some cases it even decreases. As an example consider the graph with 4000 transactions. We can calculate the karp flatt metric for varying number of cores. Refer to Table 4.1

We can observe that the karp flatt metric is remaining constant with the number of cores till 10 cores. Hence the main reason for not getting the desired speedup for increasing cores is limited opportunity for parallelism - large fraction of computation which is **inherently sequential**. The main reason is the critical section which each thread has to execute.

4.4.2 Varying number of items

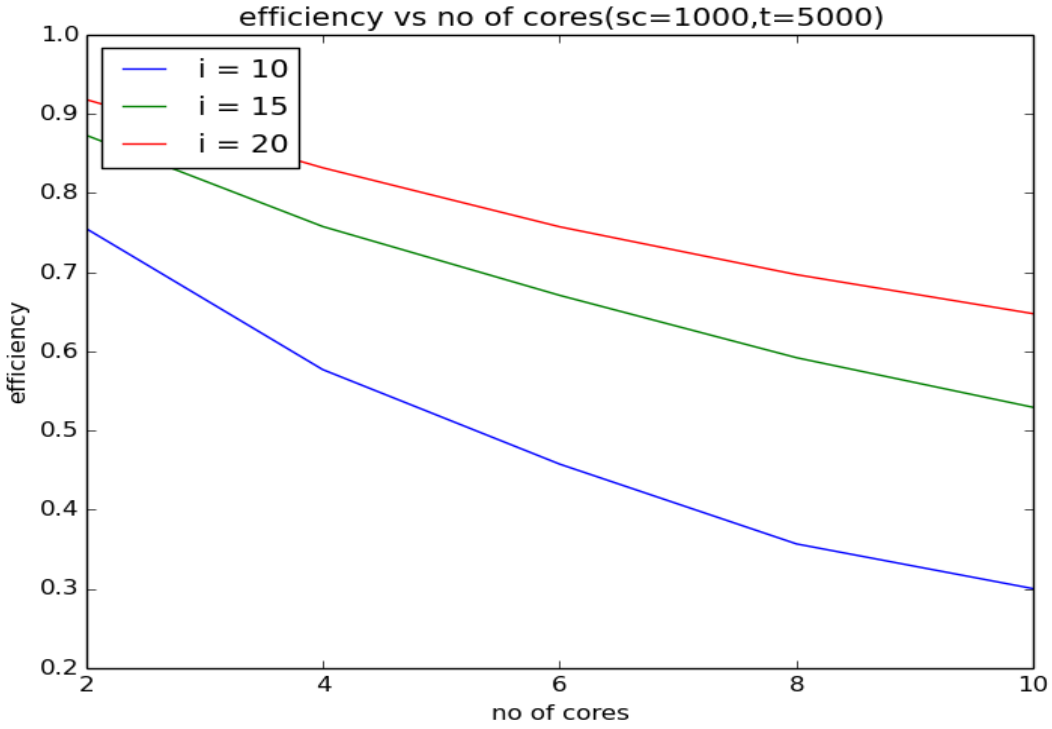


For 20 items, the speedup increases almost linearly with the number of cores. For the same number of transactions we varied the number of individual items for different graphs. This effectively means increasing the problem size. As the problem size was increased, the speedup curve saturated at larger number of processors. Consider the curve when the number of items is 20. For 20 items and 5000 transactions, the problem size is fixed. Hence we can use amdahl's law to determine the theoretical maximum speedup. By doing the gprof analysis we observed that the parallel section takes 86% of the time. Hence for infinite number of processors the speedup will be 7.142. We can observe that the speedup for 10 cores is around 6.5.

4.5 I/O optimisation

One possible optimisation is to read the entire transaction file into memory and store it in form of some list data structure. This will reduce the time required in scanning the database in both serial and parallel implementations. However this optimisation is not realistic. The size of transaction database is in millions. Due to time constraint we have restricted our analysis to small transaction sizes. So in a real life situation it would be impractical and impossible to read the entire transaction database in memory.

4.6 Efficiency vs number of cores



The efficiency is decreasing as the number of items increases. Increasing items increases the problem size. We can conclude that the problem is weakly scalable. For weakly scalable problems, the efficiency remains constant as number of processors increases by increasing the problem size.

Chapter 5

Conclusion and Future Scope

5.1 Conclusion

In this project we parallelised the apriori algorithm using OpenMP API. Initially the apriori algorithm seems difficult to parallelise because it involves loop dependencies between iterations. However we were able to exploit parallelism within each iteration. The gprof profiler provided valuable information which enabled us to select the part of code to parallelise. After parallelisation we evaluated our results to check if any more parallelisation was possible. Using the karp flatt metric, we evaluated the experimentally determined serial fraction of the parallel code for varying processors. The metric revealed that the main reason for not achieving linear speedup was the inherently serial fraction of the code and not the parallel overhead. Hence according to the metric it is not possible to further parallelise the code.

If the number of threads is increased beyond 8 threads, it was observed that the speedup did not increase proportionately. One possible reason could be because of the hardware used. The hardware specifications indicate that the cluster has 2 sockets and each socket has 8 cores. So if the number of threads is increased beyond 8, then some time will be consumed in inter socket communication. This could be a possible reason for the above observation.

5.2 Future Scope

- We have performed our analysis on a maximum of 16 cores. One possible future extension running the parallel code on large number of processors($\tilde{1}50$). This would enable us to increase the problem size(number of transactions) which better resembles a real life scenario.
- For the increased number of processors, MPI library can be used instead of OpenMP.
- For removing the critical section in the code, reduction clause can be used. However reduction for maps is not defined. From OpenMP 4.0 onwards we can define our own reduction clause. This could be used for writing the reduction clause with maps.

5.3 References

- `www2.cs.uregina.ca/~dbd/cs831/notes/itemsets/itemset_apriori.html`
- `www.cse.buffalo.edu/faculty/miller/Courses/CSE633/Velliyattikuzhi-Fall-2012-CSE633.pdf`