PARALLELIZATION OF APRIORI ALGORITHM

CS301: HIGH PERFORMANCE COMPUTING

OMKAR DAMLE
201401114
RUSHIKESH NALLA
201401106

CONTENTS

INTRODUCTION

- 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.

SERIAL PSEUDO CODE

Pass 1

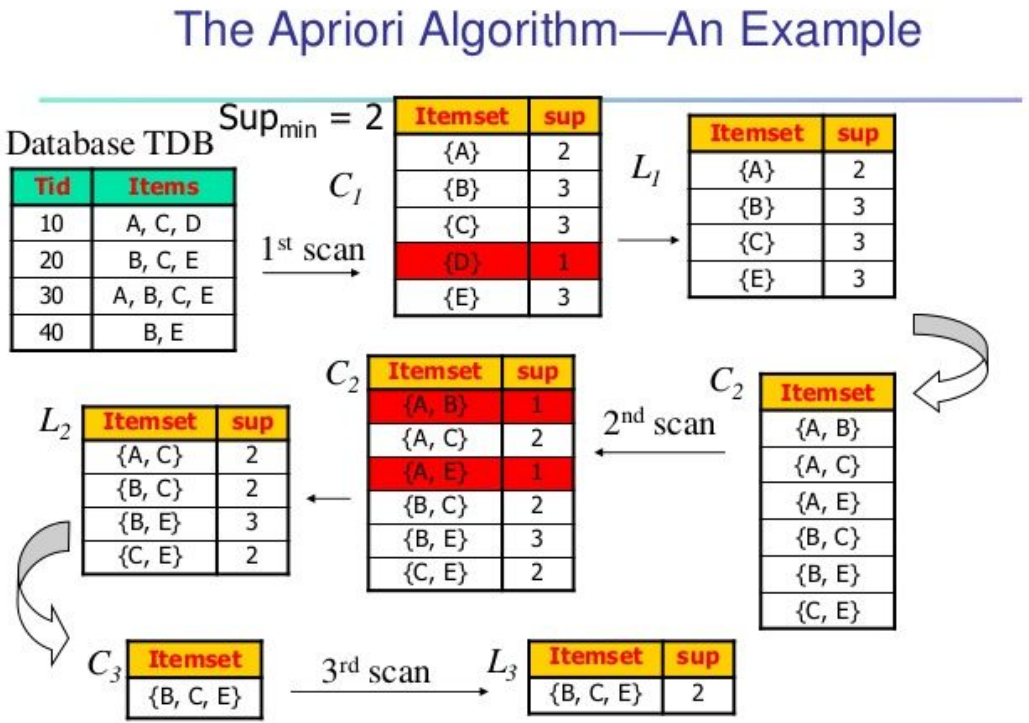
1. Generate the candidate itemsets in C1
2. Save the frequent itemsets in L1

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.item k-1, q.item k-1
    from Lk-1 p, Lk-1 q
    where p.item1 = q.item1, . . . p.item k-2 = q.item k-2, p.item k-1 < q.item k-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. Scan the transaction database to determine the support for each candidate itemset in Ck
3. Save the frequent itemsets in Lk

APRIORI EXAMPLE

- Input is the transaction database and the support count.
- 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.

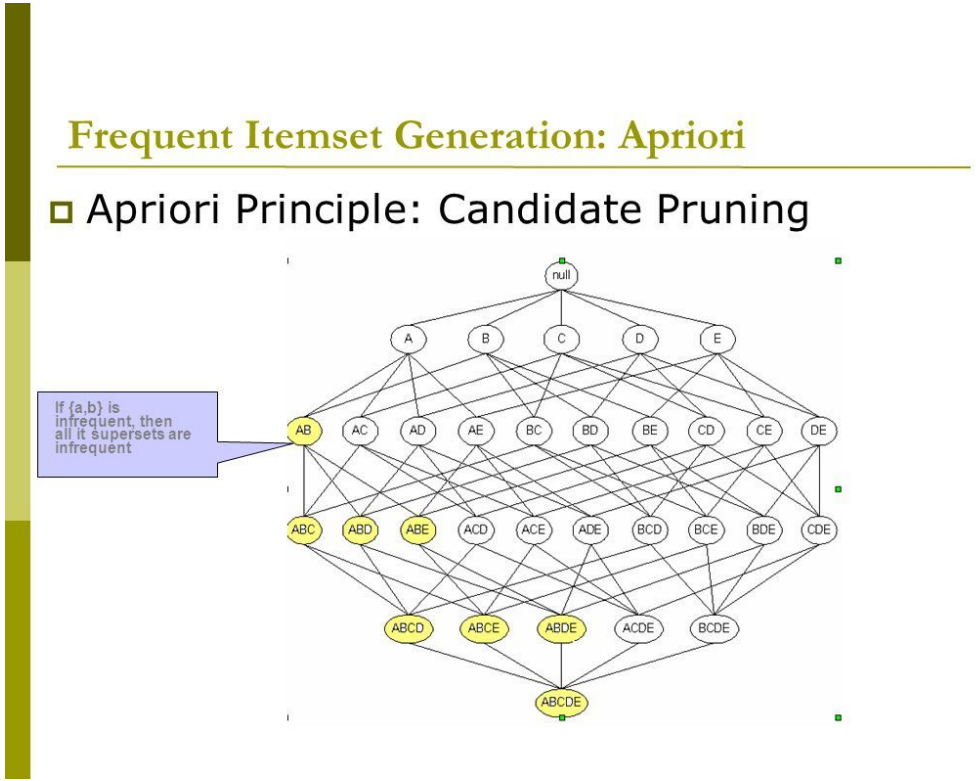


COMPUTATIONAL INTENSITY

- 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).
- Number of candidates M = 2^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.

OPTIMIZING THE SERIAL CODE

- 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.



SCOPE FOR PARALLELISATION

- **Profiling**
  - Our implementation of the apriori algorithm requires around 9 functions.
  - 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().
- **Parallelizing 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.

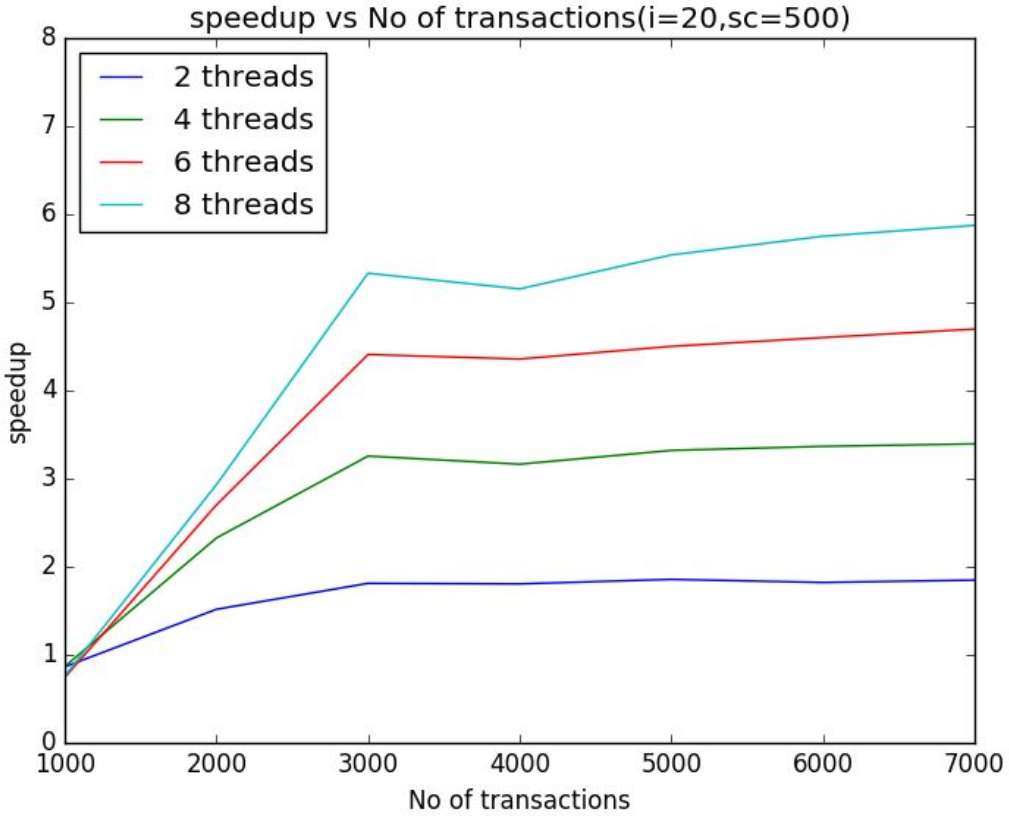
SCOPE FOR PARALLELISATION

- The transaction data was divided among the threads according to **data decomposition principle.**
- Each thread kept a local count of the candidate itemset.
- 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**.**

- **Other possibilities for parallelisation**
  - 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 bidirectional map iterators.
  - OpenMP requires iterators to be random access iterators. As a result it was not possible to parallelize these loops with openMP. A possible solution is using Intel TBB (Thread building blocks).
  - For the problem size which was feasible to run on maximum of 16 cores, other functions consumed negligible amount of time.

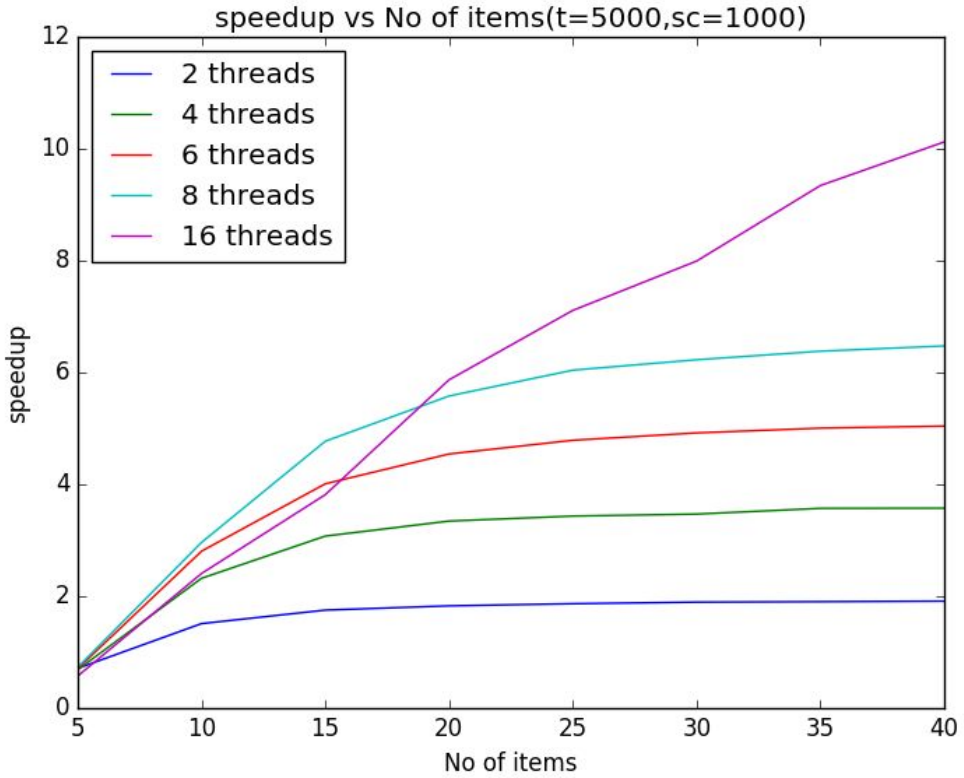
SPEEDUP CURVES

- We can see that as the number of transactions increases, speedup increases.
- The speedup saturates around 3000 transactions.
- The main reason is that the time spent in serial and parallel sections increases proportionately as the transactions increase.



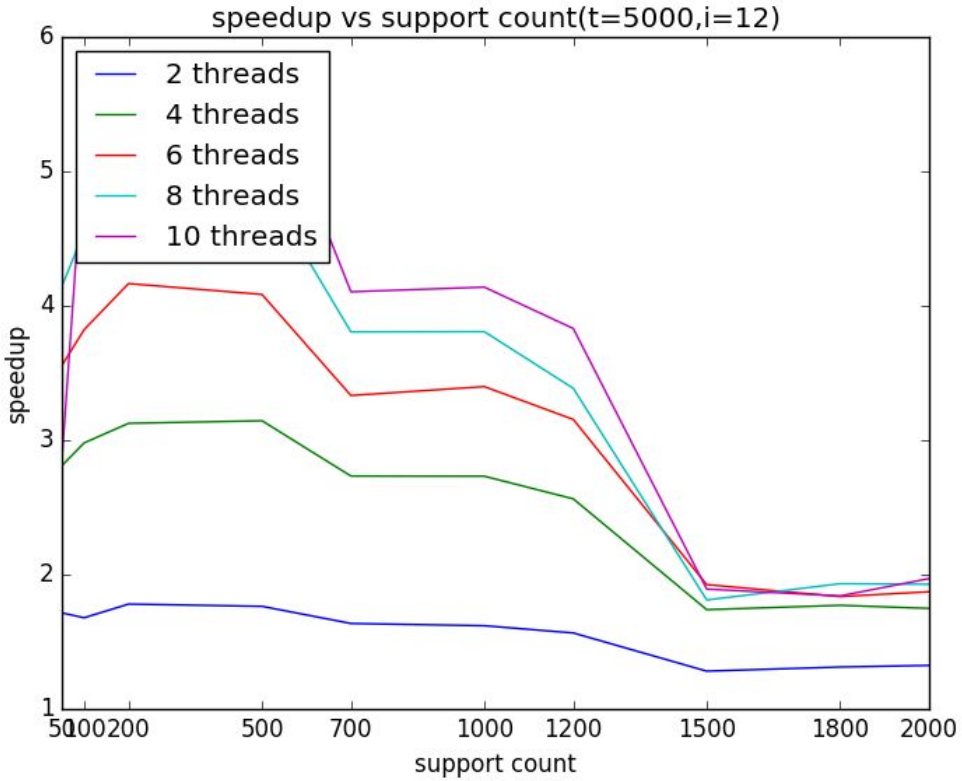
SPEEDUP CURVES

- We can see that as the number of threads 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 parallel section takes 86% of the time.
- Using 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.



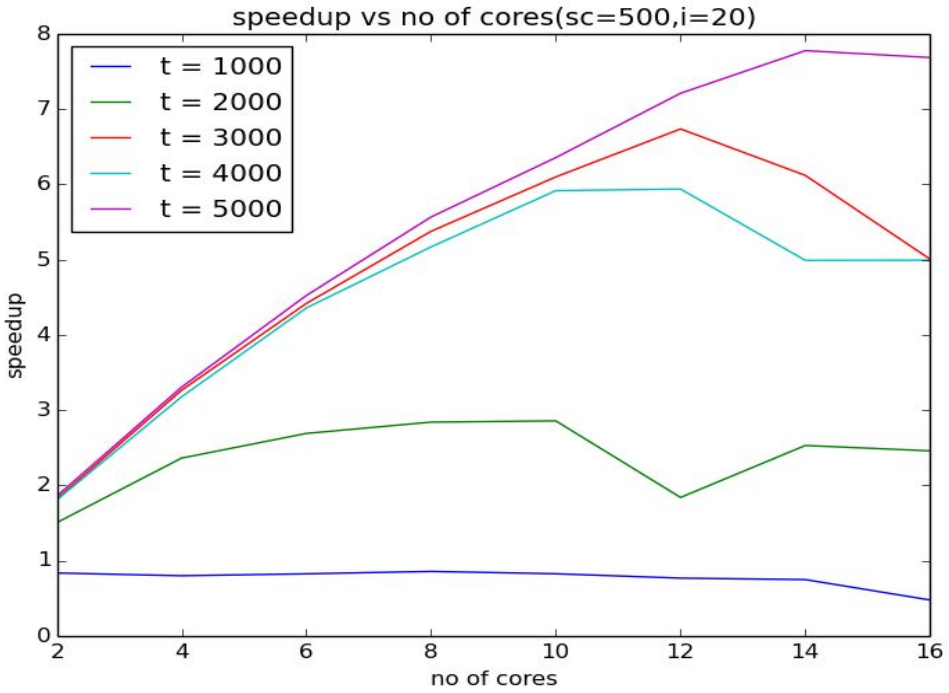
SPEEDUP CURVES

- 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 (parallel section) and hence the speedup decreases.
- We can also see that it is true for all the threads, the decreasing pattern follows.



SPEEDUP CURVES

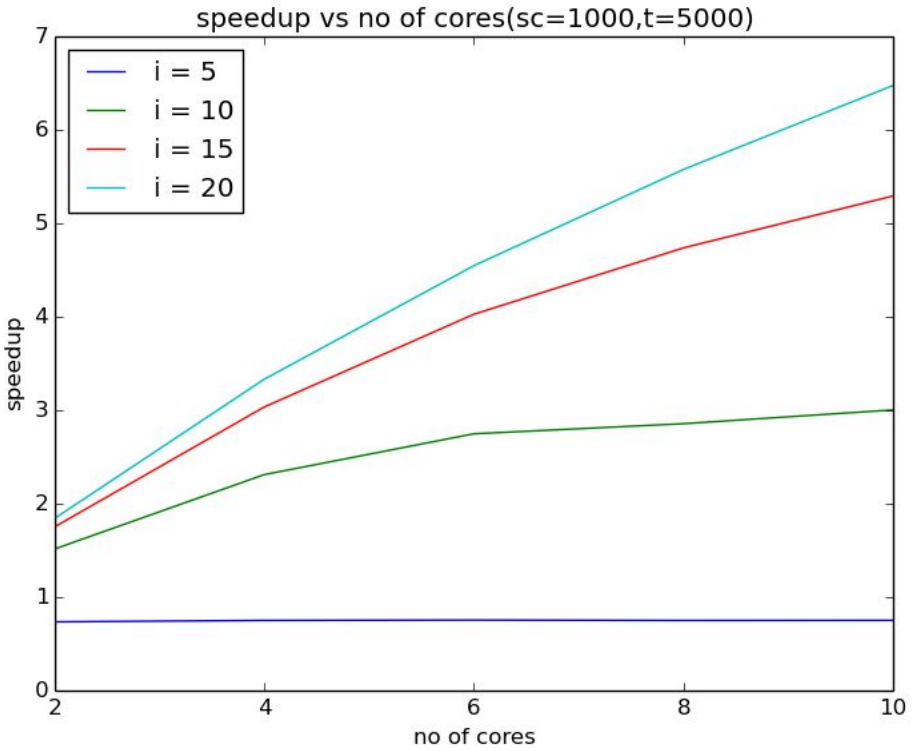
- Consider the graph with 4000 transactions. We can calculate the karp flatt metric for varying number of cores.
- Karp flatt metric is remaining **constant** with the number of cores till 10 cores.
- Hence main reason for not getting the desired speedup for increasing cores is **limited opportunity** for parallelism - large fraction of computation which is inherently sequential.
- The main reason is the critical section which each thread has to execute.



#Threads	2	4	6	8	10
Karp Flatt Metric	0.0751	0.0794	0.0756	0.0780	0.0768

SPEEDUP CURVES

- For 20 items, the speedup increases almost linearly with the number of cores.
- Varying the number of individual items for different graphs means effectively increasing the problem size.
- By doing the **gprof analysis** for 20 items and 5000 transactions we observed that the parallel section takes 86% of the time.
- We can use amdahl's law as problem size is fixed. 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.



CONCLUSION AND INFERENCES

- 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.
- For a fixed problem size, we calculated the maximum possible speedup by amdahl's law. The measured speedup was close to the theoretical speedup.
- 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.