A PROJECT REPORT ON

“PARALLELIZATION OF APRIORI ALGORITHM”

COURSE:

CSE4001 PARALLEL AND DISTRIBUTED COMPUTING

UNDER THE GUIDANCE OF **PROF. SIVA SHANMUGAM**

By

Hrishikesh Bharadwaj C – 16BCE0722

Ayush Kumar – 16BCE0374

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

ABSTRACT

Association rules are the main technique for data Mining Apriori algorithm is a conventional algorithm of association rule mining. There are many algorithms for mining association rules and their variations are proposed on basis of Apriori algorithm, but those algorithms are not efficient. With the rapid development of networks and information technology, the vast information has paid more and more attention by people. While getting the information with high speed, the analysis and mining of the information and rules hidden deep in the data are also getting more attention. Data mining technology is to organize and analyze the data, which can extract and discover knowledge from the huge and unstructured data, so how to apply the data mining techniques in transactional databases is the focus of this topic studied. In this paper, combining with parallel programming techniques, Apriori algorithm in association rule mining algorithm is described in detail, the algorithm implementation process is illustrated, and the optimized methods of the algorithm are discussed.

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.

ABOUT APRIORI ALGORITHM

Apriori algorithm is given by R. Agrawal and R. Srikant in 1994 for finding frequent itemsets in a dataset for Boolean association rule. Name of algorithm is Apriori is because it uses prior knowledge of frequent itemset properties. We apply a iterative approach or level-wise search where k-frequent itemsets are used to find k+1 itemsets. To improve the efficiency of level-wise generation of frequent itemsets an important property is used called Apriori property which helps by reducing the search space. All nonempty subset of frequent itemset must be frequent. The key concept of Apriori algorithm is its anti-monotonicity of support measure. Apriori assumes that

All subsets of a frequent itemset must be frequent(Apriori propertry).

If a itemset is infrequent all its supersets will be infrequent.

PARALLEL PROGRAMMING PARADIGMS

1. Phase parallel (Loosely Synchronous paradigm or the Agenda Paradigm) –
   - Widely used in parallel programming
   - The parallel program consists of number of supersteps, and each has two phases.
   - In the computation phase, multiple processes are there and each perform an independent computation C.
   - In the next interaction phase the processes perform one or more synchronous interaction operations, such as a barrier or a blocking communication.

- Next, the superstep is executed.
- Difficult to maintain balanced workload among processors.
- Interaction is not overlapped with computation.

2. Divide and Conquer
   - The parent process divides its workload into several smaller sub-tasks and assigns them to a number of child processes.
   - The child processes compute their workload in parallel and the results are merged by the parent.
   - Done recursively

3. Pipeline
   - Here, a number of processes form a virtual pipeline.
   - A continuous data stream is fed into the pipeline, and the processes execute at different pipeline stages simultaneously in an overlapped fashion.

4. Process farm (Master – slave paradigm)
   - A master process executes the sequential part of the parallel program and spawns a number of slave processes to execute the parallel workload.
   - When a slave finishes its workload, it sends the information to the master which assigns a new workload to the slave.
   - The master can become the bottleneck.

5. Work pool
   - This is often used in a shared variable model.
   - A pool of works is present in a global data structure.
   - Initially, there may be just one work in the pool.
   - Any free process fetches a piece of work from the pool and executes it, producing zero, one, or more new work pieces put into the pool.

GEOMETRIC DECOMPOSITION

For linear data structures like arrays, we can often reduce the problem to potentially concurrent components by decomposing the data structure into contiguous substructures, in a way analogous to dividing a geometric region into sub regions – hence it is called Geometric decomposition. For arrays, this decomposition is along one or more dimensions, and the resulting subarrays are usually called *blocks*.

This decomposition of data into blocks then implies a decomposition of the update operations into tasks. Each task represents the update of one block, and tasks execute concurrently. If each task can be executed with information entirely local to its block, then the concurrency is parallel and the Task Parallelism pattern should be used. In many cases, however, the update requires information from points in other blocks. In such a case, information must be shared between chunks to complete the update.

ALGORITHMIC PARALLELISM

Algorithms in which operations are executed step by step are called serial or sequential algorithms. Algorithms in which several operations are executed simultaneously are referred to as parallel algorithms.

A parallel algorithm for a parallel computer can be defined as set of processes that may be executed simultaneously and may communicate with each other in order to solve a given problem. "Process" may be defined as a part of a program that can be run on a processor. In designing a parallel algorithm, it is important to determine the efficiency of its use of resources available.

LEVELS OF PARALLELISM

We distinguish the following levels of parallelism:

1. **Job:** Full jobs are running completely in parallel on different processors with zero or almost zero interaction between those jobs.
   - improved throughput of computer
   - shorter real time of jobs,

2. **Program:** Parts of one program are running on multiple processors.
   - Shorter real time.

3. **Instruction:** Parallelization between phases of execution.
   - Accelerated execution of the whole instruction.

4. **Arithmetics, bit level**: Hardware inherent parallelism of integer arithmetics. Bit-wise parallel access but word-wise sequential access and vice versa.
   - Less clocks for executing an instruction.

The levels of parallelism can be combined.

PROBLEM STATEMENT:

The task that we have selected involves parallelization of the Apriori algorithm. In data mining, a classic algorithm for learning association rules is the Apriori Algorithm. Apriori is designed to operate on databases containing transactions (for example, collections of items bought by customers). The Apriori algorithm is used to group item-sets which contain a minimum number of occurrences 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.

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.

We are proposing a work of serial and parallel implementation of Apriori algorithm on a multi core processor. The proposed system tends to verify that increasing the number of cores of the

processor reduces the processing time of the algorithm by a significant amount than the algorithm being implemented on a serial manner on a single processor.

One important optimisation is in generate C () function. While generating the possible candidates at kth iteration from the frequent item sets of (k-1) st iteration, only a select combination is considered. The selection criteria are that the frequent item sets 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.

Pseudocode for Serial Algorithm:

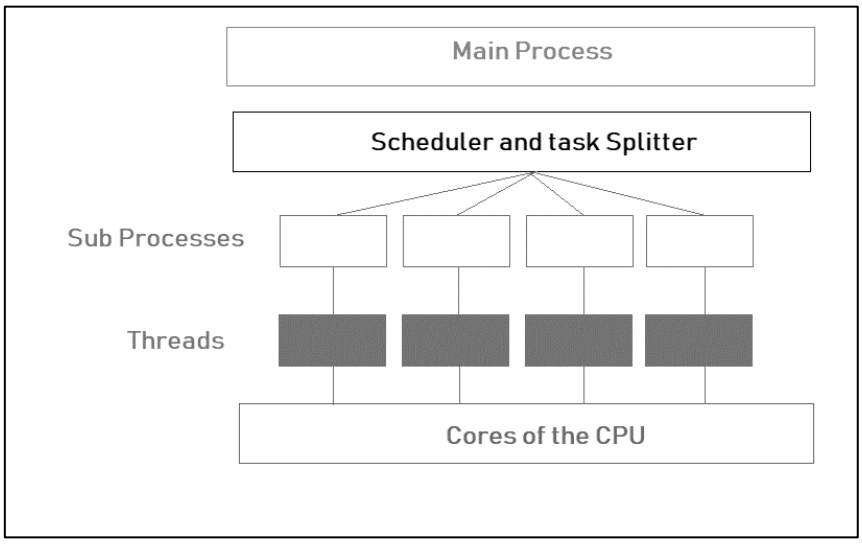
Pass 1

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
2. Join Lk-1 p with Lk-1 q, as follows:
    a. insert into Ck
    b. 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
3. Generate all (k-1) - subsets from the candidate itemsets in Ck
4. Prune all candidate itemsets from Ck where some (k-1) - subset of the candidate itemset is not in the
5. Scan the transaction database to determine the support for each candidate itemset in Ck
6. Save the frequent itemsets in Lk

PARALLEL COMPUTING ARCHITECTURE:



PARALLELIZATION OF APRIORI ALGORITHM:

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().

a. 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.

b. Other possibilities 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 open MP. A possible solution is using Intel TBB (Thread building blocks).

Parallel pseudo code

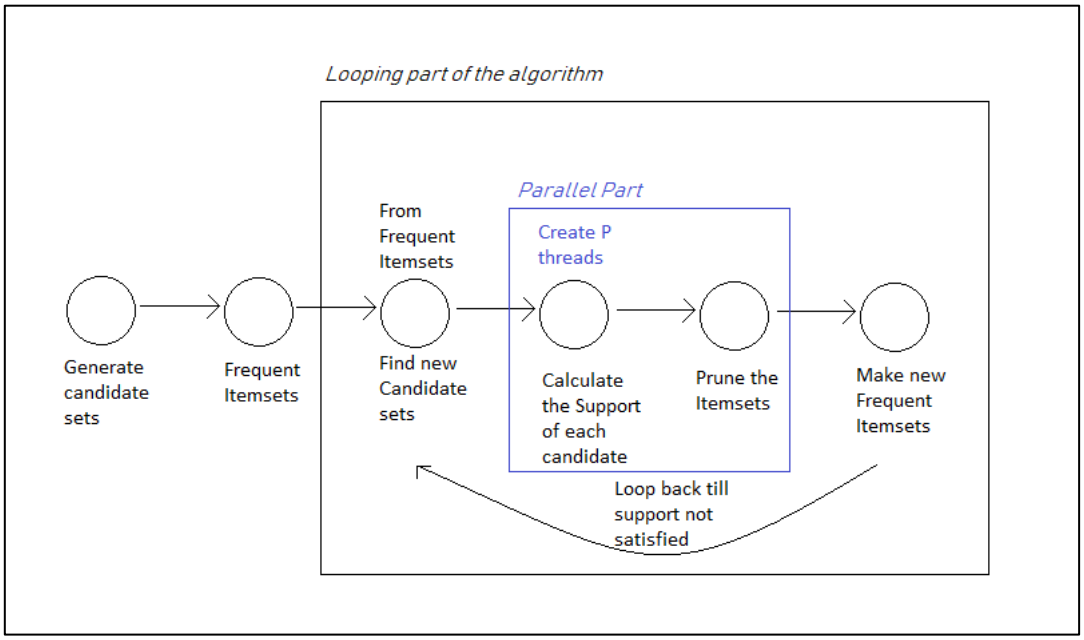
Pass 1

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

Pass k

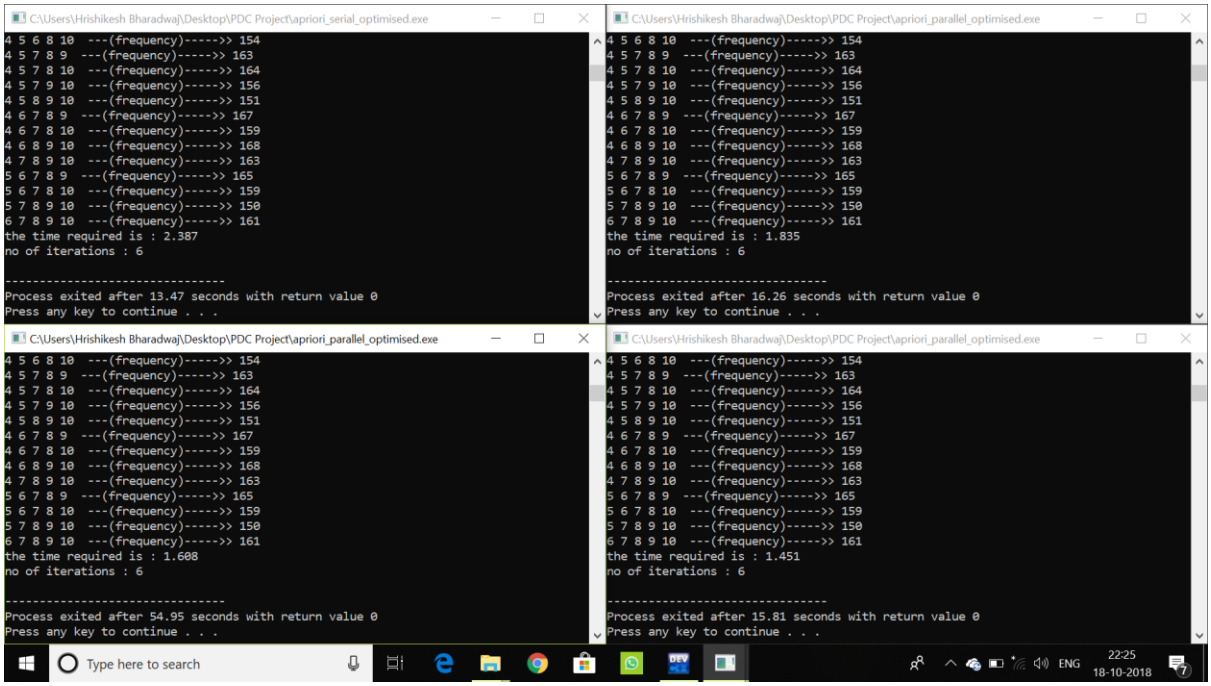
1. Generate the candidate itemsets in Ck from the frequent itemsets in Lk-1
    a. Join Lk-1 p with Lk-1 q, as follows:
    b. insert into Ck
    c. select p.item1, p.item2, . . . , p.itemk-1, q.itemk-1 from Lk-1 p, Lk-1 q
       where p.item1 = q.item1, . . . p.itemk-2 = q.itemk-2, p.itemk-1 < q.itemk-1
    d. Generate all (k-1) - subsets from the candidate itemsets in Ck
    e. 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
    a. Scan the transaction database to determine the support for each candidate itemset in Ck
3. Save the frequent itemsets in Lk

PTG REPRESENTATION:



The above diagram represents the PTG representation of how the parallel algorithm works amongst the different tasks that comprise of the main parallel algorithm of the Apriori Algorithm.

Screenshot of Output:



Comparison of Threads against serial execution, 2 threads, 4 threads and 8 threads.

```
Number of processors = 8
Number of threads = 4
Max threads = 4
In parallel? = 1
Dynamic threads enabled? = 1
Nested parallelism supported? = 0

Frequency List 5

1 2 3 8 9  ---(frequency)----->> 186
1 2 6 7 9  ---(frequency)----->> 180
the time required is : 1.368
no of iterations : 6

--------------------------------
Process exited after 17.29 seconds with return value 0
Press any key to continue . . .
```

Description of the parallel parameters.

CONCLUSION:

In this project we parallelised the Apriori algorithm using Object oriented programming. 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 the time required for the computation of the algorithm seemed to be decreasing exponentially. This clearly specifies that the computation of the algorithm becomes time efficient after parallelizing and distributing the computation within the threads of the processor.

The future scope for the parallelization of the Apriori algorithm includes:

1. We have performed our analysis on a maximum of 16 cores. One possible future extension running the parallel code on large number of processors (1-50). This would enable us to increase the problem size (number of transactions) which better resembles a real-life scenario.
2. For the increased number of processors, MPI library can be used instead of OpenMP.
3. 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.