A Comparative Study of Parallelized FP-Growth Algorithm

Chung Hsuan
IOC
NYCU
Hsinchu
yam5421363@gmail.com

Ping-Ta Tsai
IOD
NYCU
Taichung
b0938930511@gmail.com

Bo-wei Wang
IOC
NYCU
Hsinchu
blue85911360@gmail.com

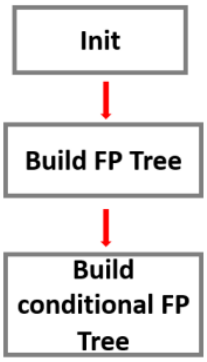
Abstract

FP-growth algorithm is a significant frequent itemset mining algorithm for discovering frequently co-occurrent items.Unfortunately,when the dataset size is huge,the computational cost can still be prohibitively expensive.In this work, we propose to parallelize the FP-Growth algorithm with OpenMP.It splits the mining task into number of independent sub-tasks, executes these sub-tasks in parallel on threads and then aggregates the results back for the final result. Experiments show that this parallel algorithm accelerate the computational speed by leveraging the dynamic tasks dispatch mechanism of OpenMP.We evaluated our algorithm on data sets with 5000 and 50000 transaction records and got the speedup of 2.79 and 2.55 .

Introduction

Nowadays, We have a broad variety of efficient algorithms for mining frequent itemsets . Agrawal et al in [1], introduced Apriori algorithm to find the frequent itemsets.The Apriori algorithm adopts candidates' generations-and-testing methodology to produce the frequent itemsets. However , in the case of the long itemsets the Apriori approach suffer from the lack of the scalability, due to the exponential increasing of the algorithm's complexity. FP-growth approach's scalable frequent patterns mining method has been proposed as an alternative to the Aprioribased approach. It is able to mine frequent itemsets without candidate generation was proposed by Han [2] . The pattern growth approach adopts the divide-and-conquer methodology to produce the frequent itemsets. We propose some approach to Parallel FP-Growth Algorithm,Which leverage strong scaling or weak scaling to accelerate computing frequent itemsets .Then , Compare and analyze the results of various methods which we proposed .

Proposed approaches



assumption:

We assume the input data with the form: {ID, items}, in which "ID" is a unique number and "items" is a set of items related to ID.

Init:

1. Find out the items and their amount we have in the whole data set.
2. Sort the items by their amount.
3. Remove items that have fewer amounts than the minimum support.
4. For each "item" related to a unique ID, sort the items by their amount.

Build FP tree:

1. For each {ID, items}, insert the items into the tree in order.
2. When inserting an item, each node visited should have their "counter" plus one.
3. Maintain a linked list for each kind of item during the construction of the tree.

Build Conditional FP tree:

1. Build a conditional FP tree for each kind of item. An item with a smaller amount should go first.
2. Find all paths from the node of the kind to the root.
3. Calculate the number of times shown in the paths for each kind of item.

4. Remove items that have fewer times shown than the minimum support. 5. Find out every possible frequent pattern rule.

[The possibility of parallel programming]

Dynamically determine whether to perform fine-grain parallelization according to the amount of data. # Only program the coarse grain parallelly, and pre-allocate many of the nodes continuously.

Sort the ID and let the sequence of inserting like depth-first (let the path be in the same pre-allocated block as much as possible)

1. init

Divise all of the data including the ID and items. # Calculate the item quantity parallelly and sort the item parallelly.

2. Construct FP tree

Classify all ID according to owning the highest of item quantity, there is no data dependency of different kinds.

How to solve the condition of skewed trees?

3. Construct conditional FP tree

The different items could construct the tree parallelly.

Experimental Methodology

we compiled our project with g++ and -O3 -m64 -fopenmp flag, and runned on the CPU intel i5-11400(6 core 12 thread) with 2.6Ghz, and we use the DDR4 32GB memory,and our data size are 5000 and 50000 transaction records.

For the algorithm, we will serial to create the biggest Fp-tree at first, and in order to find the patterns as the result we want, we will call the findPatterns function recursively, the findPatterns will check the current tree's header table, and construct others own tree by iterating the header, and result will be merge into the global variable as our final result.

Our observation show that in the biggest Fp-tree has 20000 item in header table, for the consideration of overhead, we will only run the findPatterns function in parallel while the recursion times is less than three, so after the recursion times bigger or equal than three, we will call the findPattern in serial version,i.e we will not create the more thread to avoid the number of thread is too much.

All of the patterns that each thread creates are not required to be ordered.

Experimental Result

For the correctness, we will merge the result each thread creates and compare it to serial version by using *diff* the output, but for the performance measurement, we will remove the merge step to avoid unnecessary overhead.

We will measure the speed up and efficiency for our evaluation with data size 5000 and 50000 in Figure3 and Figure 4, and plot it in Figure 1 and Figure 2. The definition of speed up is parallel execution time divides serial execution time, and the efficiency is the number of threads divides speed up. For the execution time, it consists of createFpTree time and findPatterns time, the createFpTree time could only be processed in serial, and the findPatterns part could be processed in parallel.

For the data size with 50000, figure 2 show that the efficiency is always lower than data size with 5000, the reason of the phenomenon is when the data size becomes larger, it need cost more time to createFpTree, and the proportion of that will also become larger, and the proportion which could be processed will become smaller, and the efficiency will become smaller.

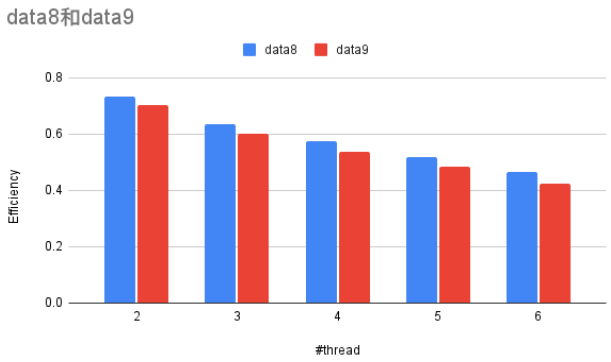


Figure1: Speed up

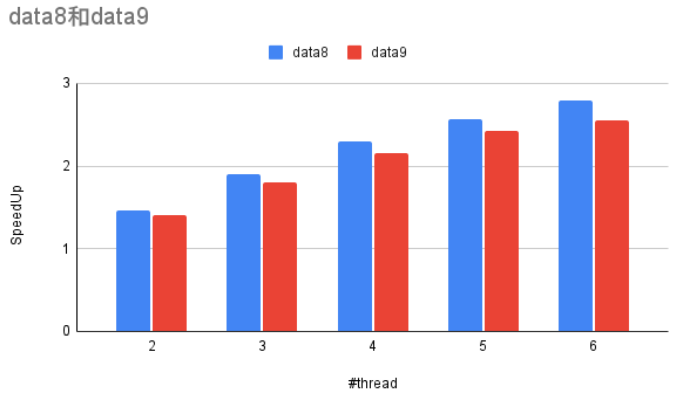


Figure2: Efficiency

	Number of thread					
	1	2	3	4	5	6
Average	0.02852	0.0194	0.01495	0.01238	0.011	0.0102
Speedup	1	1.46951	1.90653	2.30239	2.59051	2.79435
Efficiency		0.73475	0.63551	0.57559	0.5181	0.46572

Figure 3: Data size with 5000

	Number of thread					
	1	2	3	4	5	6
Average	0.02852	0.12054	0.09406	0.7879	0.07011	0.06645
Speedup	1	1.40623	1.80217	2.15159	2.41805	2.551
Efficiency		0.70311	0.60072	0.53789	0.48361	0.42516

Figure 4: Data size with 50000

Related work

Apriori algorithm : Apriori algorithm is a sequence of steps to be followed to find the most frequent itemset in the given database. This data mining technique follows the join and the prune steps iteratively until the most frequent itemset is achieved. A minimum support threshold is given in the problem or it is assumed by the user.

FP-growth algorithm : This algorithm is an improvement to the Apriori method. A frequent pattern is generated without the need for candidate generation. FP growth algorithm represents the database in the form of a tree called a frequent pattern tree or FP tree.

strong scaling and weak scaling: Amdahl's law can be formulated as : speedup = 1 / (s + p / N) . where s is the proportion of execution time spent on the serial part, p is the proportion of execution time spent on the part that can be parallelized, and N is the number of processors. Strong scaling is defined as follow . Amdahl's law states that, for a fixed problem, the upper limit of speedup is determined by the serial fraction of the code. In other words , Strong scaling is mean that , when speedup can be achieved on a multiprocessor without increasing the size of the problem . Instead , weak scaling is mean that ,when speedup is achieved on a multiprocessor by increasing the size of the problem proportionally to the increase in the number of processors .

Conclusions

In this project, we parallelize the FP-growth algorithm, which is used to find frequent patterns of different items in data mining.

We exploited the independent subtrees to generate many tasks that can be executed parallelly. Since the amount of tasks is equal to the number of items, we can get a lot of tasks to balance the load of multiprocessors. We leveraged the dynamic tasks dispatch mechanism of OpenMP to realize the design.

We evaluated our algorithm on data sets with 5000 and 50000 transaction records and got the speedup of 2.79 and 2.55 respectively on intel i5 (6 cores) with 32 GB RAM.

References

[1] Agrawal , R. , Imieliński , T. , & Swami , A."Mining association rules between sets of items in large databases".In proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, pages 207-216, Washington, DC, 1993.

[2] Han, J. , Pei, J. , & Yin, Y. "Mining frequent patterns without candidate generation". In Proc. ACM-SIGMOD Int. Conf. Management of Data (SIGMOD '96), Page 205-216, 2000.