12/1/2017

CP 431/631

Term Project – Parallel Data Mining

By group 2 (Omer Tal, Elizabeth Gorbonos, Tianran Wang)

Assignment description

The objective of the project was to implement a parallel version of the Apriori algorithm.
Apriori is a data-mining method used to discover association rules between items in a given dataset. It is an iterative algorithm where in each step *k*, a list of candidate items of size *k* are created. For each candidate itemset the algorithm calculates its support:

$$Support(x) = \frac{Frequeuncy(x)}{n}$$

Where *x* is an itemset and *n* is the number of transactions in the given dataset. Using a given minimum support parameter, Apriori then filters out itemsets with lower support than the required minimum. The iterations terminate in step *L* when the method cannot generate any *k*+1 itemset for the next iteration.
This algorithm is correct by virtue of the two properties – *downward closure* and *anti-monotonicity*. Downward closure means that a set of items included in a frequent itemset must also be frequent. On the other hand, anti-monotonicity means that infrequent itemsets cannot be contained in another frequent itemset.

Once the algorithm has found all the frequent itemsets between 2 and *L* it then can look for the rules within each itemset. This is done by generating all combinations of subset pairs (left-hand set and right-hand set, denoted *lhs* and *rhs* respectively). For each such pair a rule *r* is created, its confidence is calculated by:

$$Confidence\ (r) = \frac{support(lhs \cup rhs)}{support(lhs)}$$

and its lift is:

$$lift\ (r) = \frac{support(lhs \cup rhs)}{support(lhs) * support(rhs)}$$

The algorithm filters the rules by keeping only those with confidence higher than a given minimum confidence.

Serial algorithm design

Prior to writing the python-based parallel Apriori algorithm we set out to compare different data structures, implemented in a serial version.

1. **Matrix representation**
   We used a n*m binary matrix to represent the items in each transaction, where n is the number of transactions and m is the number of items. In this case, the value of cell i,j would be 1 if item j was part of transaction i and 0 otherwise. The benefit of such representation is that the complexity of checking the existence of a specific item in a transaction is O(1). However, due to the sparse nature of the datasets we tested, this representation requires a high amount of memory and takes O(n*m) to compute the support of the candidates. Therefore, we found this method to be inefficient compared to other approaches.

2. **Vertical data format**
   In this version we transformed the data to a list of itemset (first, each with only one item), where each itemset links to a sorted list of transactions it is part of. Then, in order to generate the list of

candidates we compute the union of two itemsets and count their matching transactions as the combined itemset's support. While the benefits of such data structure was improved performance when generating the candidates and computing support, its drawback was the necessary preprocessing which resulted in inferior performance compared to our next approach.

3. **Transaction list**

   For this method we stored all of the transactions in a list. The items in each transaction were represented by a dictionary to allow an O(1) access. This approach included few additional optimizations such as constructing an itemset tree to prune the candidate space and removing transactions once they are deemed obsolete.
   For each step *k* > 1, we build an itemset tree from a list of next step candidates and used it to count the frequencies of the candidates in the dataset. Each transaction which contained less than *k*+1 itemsets was removed since it cannot benefit any next iterations. This method has a significantly lower memory requirement than the first method and is efficient due to the data structures it employs.

   Considering its advantages, superior runtime and relatively simple parallel decomposition we decided to choose the third approach as the base for our parallel algorithm design.

Parallel algorithm design

We considered two load balancing paradigms – static and dynamic load balancing. The static approach consisted of dividing the workload before running the program, based on the number of transactions and processors. The dynamic approach involved a designated master which assigns partitions of transactions to slave processors. After processing each partition, the slaves return the results to the master and request another partition to work on.

1. **Static load balancing**

   Implementing this paradigm, each process *p* loads the whole dataset and extracts its relevant n_P transactions, which is determined by the formula:

$$n_p = \left\lfloor \frac{n}{P} \right\rfloor + \begin{cases} 1 & if\ p < mod(n, P) \\ & 0\ else \end{cases}$$

   Each processor computes the transactions ranging from $i_{start,p}$ to $i_{start,p} + n_P$ where:

$$i_{\ start,p} = p * \left\lfloor \frac{n}{P} \right\rfloor + min(p, mod(n, P))$$

   After counting the itemset frequencies in the process work range, each processor sends the results to process 0, which then prepares the next level candidates and the itemset tree. This tree is then broadcasted to all other processes and is used for the next iteration. At the last iteration process 0 broadcasts None as a stop signal and generates the association rules. Although, generating the rules can be parallelized, we deemed it was not worth it as it took a very short time (less than a second for the tested datasets).

2. **Dynamic load balancing**

   Unlike the previous design, in this version only the master process loads the dataset. It then splits it

into *r* partitions, each of size $log_2(n)$. Each slave process requests a task from the master process, the master assigns it the next available partition and sends all the associated transactions using the MPI send command. As part of processing the partition, the slave deletes unnecessary transactions (with infrequent items) and returns the reduced list of transactions to the master along with its frequency results. As in the previous design, after all partitions were processed, the master process filters the infrequent itemsets and generates both the itemset candidates and the itemset tree, which is then broadcasted to all other processes.
As may be inferred, this method relies more on communication than the static approach.
The comparison of the two methods will be described in the results section.

Executing the program

The program is meant to run using python 3.4.2, and can be executed using the following parameters:

1. **Arguments:**
   a. `-d DATASET, --dataset DATASET`
      the name of the dataset file located in the *datasets* folder.
      The dataset is expected to be of the format:
      item1 item2 item3
      item1 item4
      Each transaction should be in its own line and items are should be delimited by a white space.

   b. `-s SUPPORT, --support SUPPORT`
      `minimum support for frequent item sets (float)`
   c. `-c CONFIDENCE, --confidence CONFIDENCE`
      `minimum confidence for association rules (float)`
   d. `-m MODE, --mode MODE`
      `load balancing mode. May be static or dynamic`
   e. `-v, --verbose`
      `include verbose debug output`

2. **Example**:
   a. **Local run**: `mpirun -np 4 python3 parallel-apriori.py --confidence=0.6 --support=0.01 --dataset=ds1.txt --mode=static`
   b. **Sharcnet queue**: sqsub -r 120m -o output.log -q mpi -n 4 python3 parallel-apriori.py --dataset=ds1.txt --confidence=0.6 --support=0.01 --mode=static

Datasets

We used five datasets to evaluate our program. We present their statistics here:

Dataset	No. transactions	No. unique items	Density	Source
DS1	88,162	16,470	0.0626%	retail.dat
DS2	25,900	4,071	0.5140%	Online Retail
DS3	119,578	1,728	0.3957%	Ta-feng dataset
DS4	4,627	111	16.6983%	supermarket.arff

DS5	1,057,944	16,470	0.0626%	

DS4 is Weka's built in supermarket dataset and DS5 is a liner expansion of dataset DS1 (we have duplicated the transaction 12 times).

Ds1 and DS5 were both originally of a compatible format. However, all other databases required a preprocessing stage to adjust them. This was done using the scripts in the tools directory (online_retail_ds_converter.py, ta_feng_ds_converter.py, weka_ds_converter.py)

Results

Parameters and Verification

In general, we aimed to mine at least 4 levels of frequent itemset. The support parameter was tuned to achieve this goal. As mentioned before, the rules generation is done in serial for all cases. Therefore, the confidence parameter had no effect on speedup and was chosen arbitrarily such that we retrieve a large (but not too-large) number of rules.

The parameters were set as follows:

Dataset	**Support**	**Confidence**
DS1	0.01	0.6
DS2	0.01	0.6
DS3	0.001	0.3
DS4	0.15	0.85
DS5	0.01	0.6

The numbers of frequent itemset for each level is summarized below:

Dataset	**No. levels**	**No. Frequent item sets**	**No. Rules**
DS1	4	Level 1 - 70 frequent itemsets Level 2 - 58 frequent itemsets Level 3 - 25 frequent itemsets Level 4 - 6 frequent itemsets	90
DS2	4	Level 1 - 598 frequent itemsets Level 2 - 404 frequent itemsets Level 3 - 82 frequent itemsets Level 4 - 3 frequent itemsets	195
DS3	4	Level 1 - 371 frequent itemsets Level 2 - 7624 frequent itemsets Level 3 - 13815 frequent itemsets Level 4 - 14 frequent itemsets	117
DS4	6	Level 1 - 39 frequent itemsets Level 2 - 346 frequent itemsets Level 3 - 947 frequent itemsets Level 4 - 767 frequent itemsets	370

		Level 5 - 153 frequent itemsets Level 6 - 2 frequent itemsets	
DS5	Same results as DS1		

To verify our results, we compared the frequent itemset and rules generated by our program to the results of an Apriori implementation by Christian Borgelt. The executable and documentation can be found here: http://www.borgelt.net/doc/apriori/apriori.html.

We developed a verifier script (tools/verifier.py) to perform this task.

Benchmarks

We present the benchmarks obtained for each dataset along with speedup and Karp-Flatt metric calculation. Speedup is calculated using the following formula:

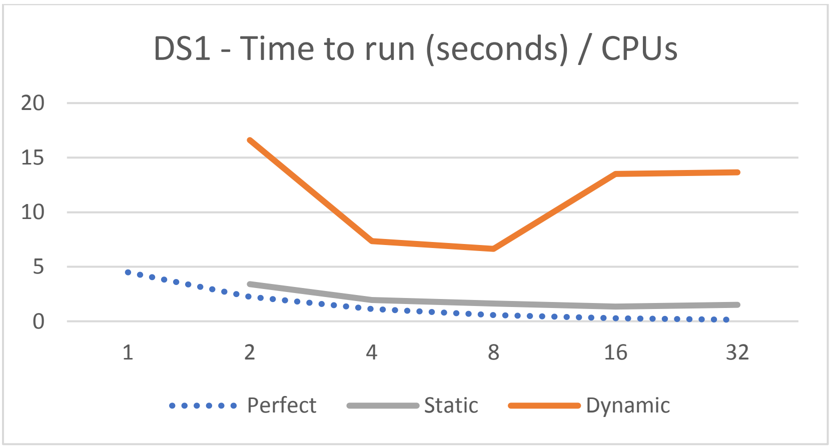
$$\psi = \frac{T(1)}{T(p)}.$$

And the Karp-Flatt value is calculated according to:

$$e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}}$$

1. **DS1**

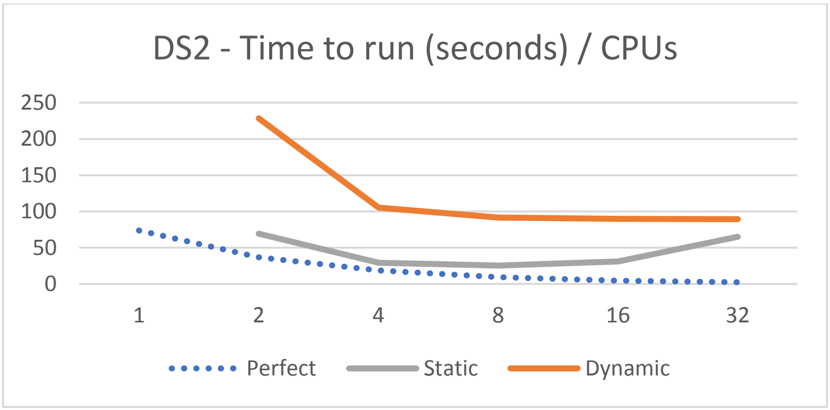
Number of CPUs	**Time dynamic LB (seconds)**	**Time Static LB (seconds)**	**Perfect Speed(seconds)**
1	4.490	4.490	4.490
2	16.606	3.405	2.245
4	7.347	1.944	1.123
8	6.638	1.634	0.561
16	13.499	1.350	0.281
32	13.660	1.512	0.140



Number of CPUs	Speedup dynamic	Karp-Flatt dynamic	Speedup static	Karp-Flatt static
2	0.27	6.40	1.32	0.52
4	0.61	1.85	2.31	0.24
8	0.68	1.55	2.75	0.27
16	0.33	3.14	3.33	0.25
32	0.33	3.11	2.97	0.32

2. DS2

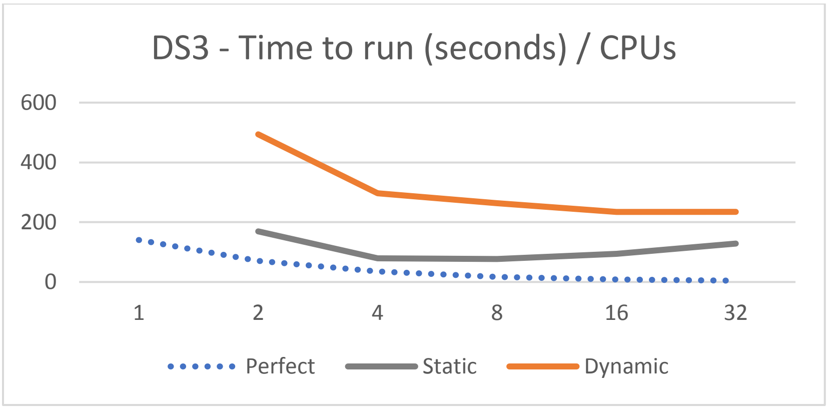
Number of CPUs	Time dynamic LB (seconds)	Time Static LB (seconds)	Perfect Speed(seconds)
1	73.727	73.727	73.727
2	228.405	69.294	36.864
4	105.155	29.292	18.432
8	91.532	25.352	9.216
16	89.706	31.193	4.608
32	89.325	65.150	2.304



Number of CPUs	Speedup dynamic	Karp-Flatt dynamic	Speedup static	Karp-Flatt static
2	0.32	5.20	1.06	0.88
4	0.70	1.57	2.52	0.20
8	0.81	1.28	2.91	0.25
16	0.82	1.23	2.36	0.38
32	0.83	1.22	1.13	0.88

3. DS3

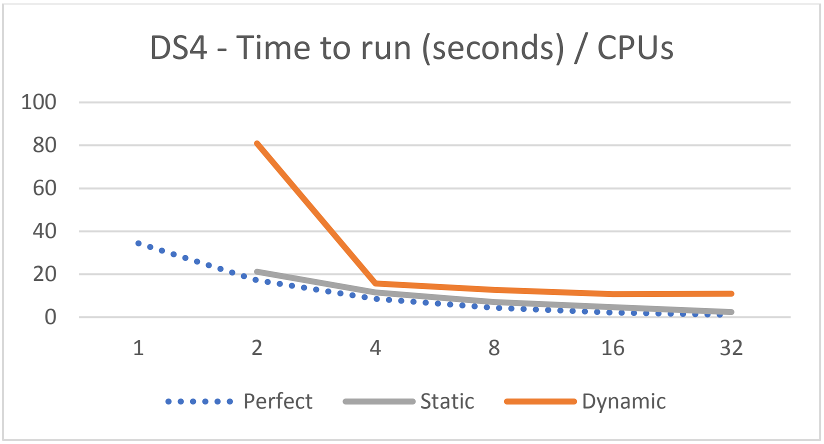
Number of CPUs	Time dynamic LB (seconds)	Time Static LB (seconds)	Perfect Speed(seconds)
1	140.490	140.490	140.490
2	493.904	169.316	70.245
4	297.015	78.968	35.123
8	263.017	76.875	17.561
16	234.290	94.164	8.781
32	234.872	128.644	4.390



Number of CPUs	Speedup dynamic	Karp-Flatt dynamic	Speedup static	Karp-Flatt static
2	0.28	6.03	0.83	1.41
4	0.47	2.49	1.78	0.42
8	0.53	2.00	1.83	0.48
16	0.60	1.71	1.49	0.65
32	0.60	1.69	1.09	0.91

4. DS4

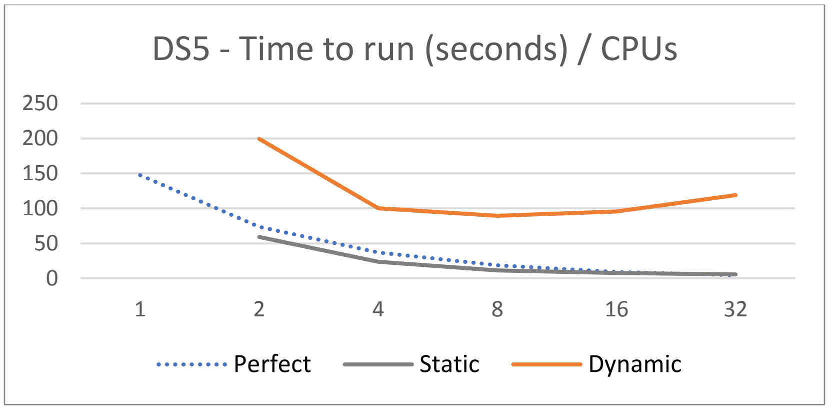
Number of CPUs	Time dynamic LB (seconds)	Time Static LB (seconds)	Perfect Speed(seconds)
1	34.387	34.387	34.387
2	80.918	21.150	17.194
4	15.703	11.549	8.597
8	12.740	7.024	4.298
16	10.725	4.589	2.149
32	10.889	2.378	1.075



Number of CPUs	Speedup dynamic	Karp-Flatt dynamic	Speedup static	Karp-Flatt static
2	0.42	3.71	1.626	0.201
4	2.19	0.28	2.977	4.066
8	2.70	0.28	4.896	4.279
16	3.21	0.27	7.493	5.762
32	3.16	0.29	14.463	4.626

5. DS5

Number of CPUs	Time dynamic LB (seconds)	Time Static LB (seconds)	Perfect Speed(seconds)
1	147.579	147.579	147.579
2	199.433	59.172	73.790
4	100.217	23.671	36.895
8	89.445	11.454	18.447
16	95.633	7.484	9.224
32	118.921	5.589	4.612



Number of CPUs	Speedup dynamic	Karp-Flatt dynamic	Speedup static	Karp-Flatt static
2	0.74	1.70	2.49	-0.20
4	1.47	0.57	6.23	-0.12
8	1.65	0.55	12.88	-0.05
16	1.54	0.62	19.72	-0.01
32	1.24	0.80	26.40	0.01

Conclusions

From the reported results we can easily see the static load-balancing version exceeded the speedup of the dynamic load-balancing one.

One interesting observation is that for DS1, DS2 and D3 the dynamic load-balancing parallel program caused in fact a slow-down in comparison with the serial program. We believe this effect occurs due to sparsity and relatively small size of those datasets. To test this assumption, we generated DS5, which has the same characteristics as DS1 but 12 times more records. Running the dynamic load-balancing version on this dataset obtained a speedup of 1.6. Additionally, we note that for DS4 we were able to achieve a speedup of 3.2. Another possible improvement for the dynamic version would be using the numpy-based communication methods (Send and Recv). To do so, the data will be represented differently (for example, translating alpha-numeric item keys to numeric).

Analyzing the Karp-Flatt values we can see that for the first 3 datasets the measures for the static load-balancing methods start increasing after 4 CPUs, meaning the parallel overhead becomes evident. For the dynamic-load balancing we see this trend emerge at higher CPU values, for instance at 16 CPUs for DS1 and DS4 and at 8 CPUs for DS5.

Another intriguing result is the super-linear speedup the static-load balancing approach produced for dataset DS5. We presume this an example of better memory utilization in parallelization.