The Grand Rule Problem

High Performance Computing

Igor Nelson Garrido Da Cruz
Faculdade de Ciências e Tecnologia
Universidade de Coimbra
Portugal
igorcruz@student.dei.uc.pt

Francisco Javier Farfán Contreras
Faculdade de Ciências e Tecnologia
Universidade de Coimbra
Portugal
contreras@student.dei.uc.pt

I. Introduction

With this report we intend to transmit and analyze the results obtained for the Grand Rule Problem.

The Grand Rule Problem was proposed and written by Professor Paulo Marques.

An industrial client (a bank) has recently approached a company in our town to classify the expenses of its clients according to a number of rules.

In practice, for each bank transaction of an input file, they want to see if it matches a number of predefined rules.

A bank transaction is a tuple with 10 attributes – $(a_0, a_1, a_2, a_3, \ldots, a_9)$, where each attribute is an integer (32 bit). Each rule is also a tuple with 10 attributes plus a classification "c". I.e., $(r_0, r_1, r_2, r_3, \ldots, r_9, c)$. Each rule attribute can either be a number (32 bit), or a "*", which means "don't care". For example, the transaction (1,2,3,4,5,6,7,8,9,0) matches the rule (1,2,3,4,5,6,7,8,9,0,111) and the rule (1,*,*,*,*,*,*,*,*,0,222), but it does not match either the rules (2,2,3,4,5,6,7,8,9,0,333) or (2,*,*,*,*,*,*,*,*,0,444).

Our challenge is to implement a system that allows efficiently solving this problem for large amounts of data. In particular, the objective is to process 10 input file containing the transaction of the last 10 days. Each input file has 1M tuples. The rule table is fixed and has 2M rules. Most of the rules are actually empty (i.e., with"*"), having numbers in only a few columns of each rule.

II. Description of the Environment Tests

In order to test quality of our algorithm we used two different machines. The first machine was used until we have a fast enough solution to advance to the tests in the final machine.

Our test machine (test environment number 1):
- CPU: 2.53 GHz Dual Core
- Memory: 4GB
- SO: Windows

Final machine (test environment number 2):
- CPU: Core 2 Duo 2.5GHz
- Memory: 4GB
- SO: Linux

III. Description of the Best Solution

A. Simple Approach

In order to solve this problem we started with a simple approach that stored all the 2M rules in memory and for each new input line verified all the possible matches with all the rules.

This approach scored 23 transactions per second in the environment number 1.

B. Sorting and Indexing

After this naive implementation we decided to sort the rules in order to store the indexes for a fast verification.

We used quick-sort algorithm and stored the indexes of the first two columns. This way we need only to verify the indexes of the rules table that start with the two numbers that we read and the ones that start with combinations of these numbers and zero.

This approach scored 91 transactions per second in the environment number 1.

C. AVL Tree

The results were not going good and we decided to try an AVL tree that stores one rule per node.

The concept idea was that we only need to descend down the tree until we find a rule node which is bigger than the input that we are searching for.

Unfortunately the results were not so great. This happened because the number of rules is too big what causes the tree to grow immensely and the pointer s to the nodes causes big overhead.

This approach scored 12 transactions per second in the environment number 1.

D. Tree of rule Indexes

In this approach we decided to use each node of the tree to store the links to the next node in an ArrayList or HashMap depending on the number of links (this study will be shown in the next section). We also changed to Java for faster implementation proposes.

This approach scored 2200 transactions per second in the environment number 1. In the test environment number two It scored 20000 without impression of output.

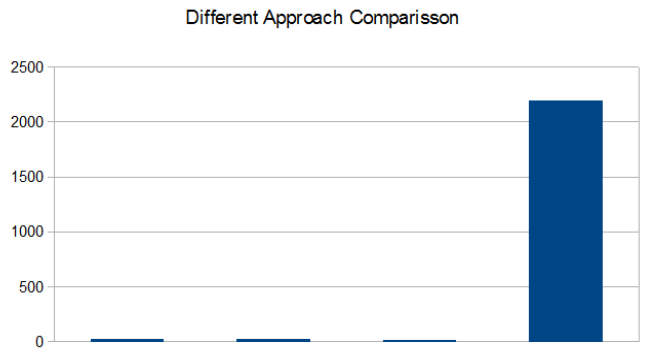

Figure 1 - Different Approach Plot

After trying all these implementations we decided to chose the last one since it gave far better results.

In more detail, we have a Graph, each node represents a single value of a given rule and points to all the possible values that can be achieved if this value occurs. All these values are stored in ArrayList if they are less than a given index or in a HashMap if their number is greater than that index.

IV. Parameter Optimization

So after we chose what algorithm to use, we had to optimize its parameters in order to attain the maximum performance. For this we changed the index that transforms the ArrayList of each node to a HashMap.

All the following were measured without printing the output in the test environment number 2.

Index	Transactions/Sec
5	7253
10	7252
15	9470
17	9412
20	7365
25	7235

Figure 2 – Changing the Transformation Index

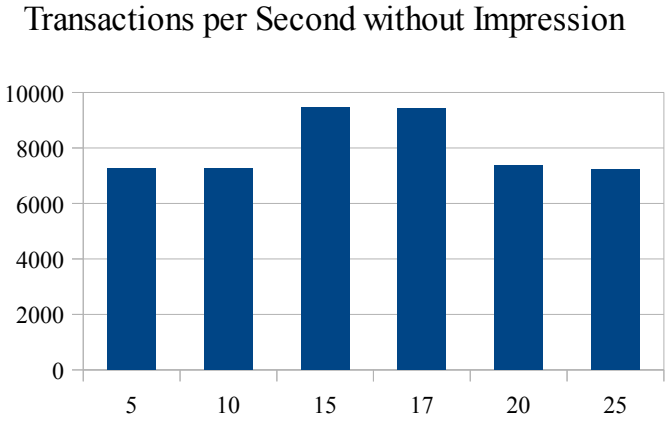

Figure 3 – Changing the Transformation Index

After achieving these results we decided to stick with the index 15 for all the following tests. Since our experiences gave better results with this index.

V. Multi Thread Scheme

To increase the performance of our algorithm we start to use a single multi-thread scheme. Having a master thread that reads inputs and sends them to an ArrayBlockingQueue while the consumer threads take work from that ArrayBlockingQueue to analyze was our first experience. However the performance was really bad.

After this implementation we decided to let the main thread help the worker threads consuming also elements from the blocking queue.

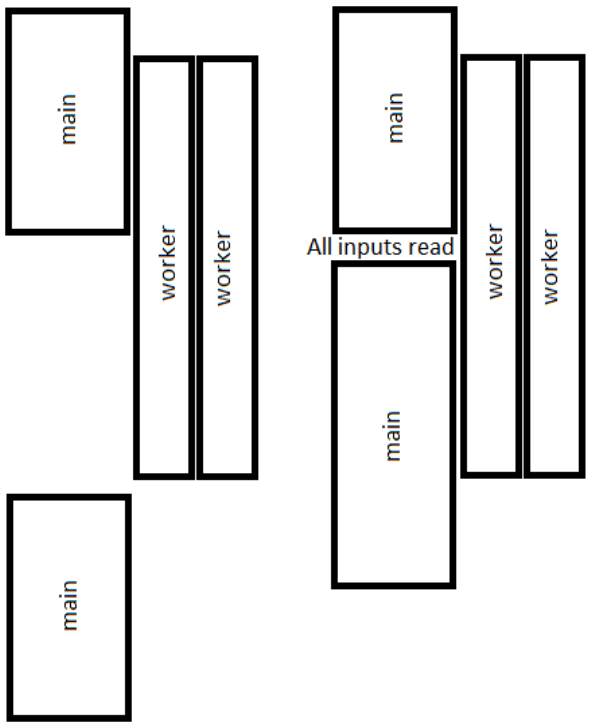

Figure 4 – In the left The first implementation, in the right the second implementation

First Implementation : 15000 ~trans/sec
Second Implementation : ~22000 ~trans/sec

Our results were far better with the second implementation.

VI. Analysis of performance Of Different Number of Threads

To test the performance with different numbers of threads we calculated the average amount of time (in seconds) after 5 runs with each amount of threads.

NThread\Time	Run 1	Run 2	Run 3	Run 4	Run 5	Average (s)
1	58	59	54	59	55	57
2	47	45	46	46	51	47
3	53	51	52	50	54	52
4	47	51	50	51	48	49,4
5	51	59	54	56	54	54,8
6	62	60	54	61	54	58,2
7	55	54	58	62	53	56,4
8	54	54	54	57	57	55,2
9	51	53	62	50	84	60
10	61	61	60	69	59	62

Figure 5 – Time Consumed per Number of Threads

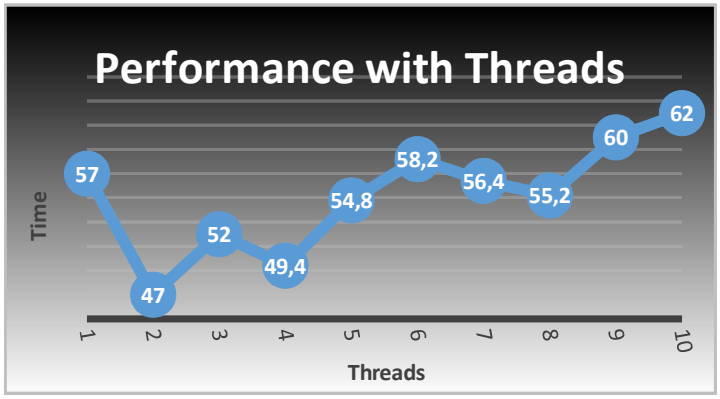

Figure 6 – Plot showing Time Consumed per Number of Threads

As we can see from the information above the best performance was achieved with 2 threads (~21200 trans/seg). However using 4 threads was not bad. When we start adding more threads we can see that the performance starts slowly decrease.

VII. How to Run

In order to run our.jar file we use java -Xmx4g -jar final.jar to set the max amount of heap size of the Java Virtual machine to 4GB since we need a large amount of memory to store the rules tree structure.

VIII. Diverse Input Files

With others input files the time does not chance much. In the next table we show the values obtained.

Figure 6 – Time per file

File	0	1	2	3	4	5	6	7	8	9
Time(s)	46	46	51	54	53	55	53	54	49	59

As we can see the time that each file took to process did not vary much. In average it was 53s which gives ~18800 transactions per second.

IX. Conclusions

Finally to conclude, we can say that for the realization of such large-scale projects need to have a large knowledge in programming, algorithms and inner workings of the computer.

After experiencing multiple algorithms and make diverse optimizations we can achieve a way to perform big inputs in a fast way.

The exercise we had to solve was a big problem considering the size of the files we had to process, and not only that, you need to process a large number of input in a short time, namely, it was not enough to make a design that the program perform its function well and will take too long time to execute.

After experiencing diverse optimizations and numbers of threads we can really achieve a fast algorithm.