Parallel Logic Regression on High Dimensional Boolean Space

CHEN-HO CHEN*
jhchen861225@gmail.com
National Yang Ming Chiao Tung University
Taiwan

HAO-JU CHANG*
black1120rock@gmail.com
National Yang Ming Chiao Tung University
Taiwan

TING-YUN HSIAO*
tina19990717@gmail.com
National Yang Ming Chiao Tung University
Taiwan

ABSTRACT

Logic regression can do an approximate reconstruction of unknown systems and is usually used to analyze a black-box circuit. In the 2019 ICCAD CAD competition, there is a topic of Logic regression and will give a black-box circuit, and then the contestants have to find the minimum approximate Boolean logic of this circuit. With a large number of circuit inputs, we need a more efficient method to solve the question. We use OpenMP in each step to speedup the Logic regression. The final result shows that with Parallelization techniques, we achieved 4.5X speedup of the Logic regression.

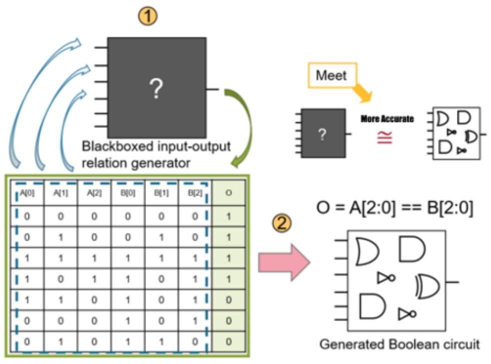
1 INTRODUCTION

Logic regression on Boolean space aims to find a Boolean-based logic circuit that satisfies the given pattern with multiple sets of inputs and outputs. With logic regression, people could approximately reconstruct the unknown system. This technology can help analyze a black-box model and could be applied to different areas of IC design, such as testing, verification, data analysis, and so on.

In 2019, a problem of logic regression on high dimensional Boolean space appeared in the ICCAD CAD Contest [5]. Contestants were required to find a minimal Boolean logic circuit which matches the inputs and outputs given by the generator. However, even with the effective algorithms devised by contestants, there still exist some special cases which are unacceptable to the algorithms and eventually have no choice but to be solved by the exhaustive method. The exhaustive method using Quine-McCluskey Algorithm[2][4], which requires a huge number of calculations and has the possibility of being parallelized. Therefore, our purpose is to parallelize the exhaustive method, and accelerate the programming process to solve the logic regression problem.

In IC design, it will cost a lot of time in testing, verification, and data analysis, the modern IC even has 1M(10^6) logic gates. Thus, it is necessary to minimize the circuit size to reduce the cost of the time. In this problem, the circuit size is defined as the number of 2-input primitive gates. The circuits we are considering in this problem the more complex circuits cannot guess the circuit behavior through the input-output relation(like an adder, multiplier). The number of the circuit input variables ranges from 25 to hundreds, and the number of circuit output variables is less than 500.

Black-box(Fig. 1) is an input-output golden circuit relation generator that can generate golden output by given input testing pattern. We can use the input-output relation which is generated by the black-box to generate an approximate circuit. The accuracy of the approximate circuit is defined as $\frac{|CorrectResult|}{|TestingPattern|} \times 100\%$. $|CorrectResult|$ is the number of the golden circuit result is the same as the approximate circuit result under the same testing pattern. $|TestingPattern|$ is the number of the testing pattern(we will give 100 thousand testing patterns in this problem). The goal of the problem is to generate an approximate minimize circuit under the accuracy constraint.


1
?
Blackboxed input-output relation generator
Meet
?
More Accurate
≅
2
O = A[2:0] == B[2:0]
Generated Boolean circuit

A[0]	A[1]	A[2]	B[0]	B[1]	B[2]	O
0	0	0	0	0	0	1
0	1	0	0	1	0	1
1	1	1	1	1	1	1
1	0	1	1	0	1	1
1	1	0	1	0	1	0
0	0	0	1	0	1	0
0	1	0	1	1	0	0

Figure 1: Problem statement with Black Box Circuit

2 PROPOSED SOLUTION

First, Tree-base Circuit Modeling will adopt to construct the accurate satisfied circuit by implicants.In order to reduce the cost in digital circuit design, using a minimum number of logic gates to implement a logic function would be desired. Given an incompletely specified function(with don't-care terms), we need to find all the prime implicants first and then select a set of prime implicants with minimum number of literals to cover all the minterms in the original logic function. In the most common way, Quine-McCluskey would be applied to the first step and Petrick's Method would be applied subsequently as shown in Fig. 2.

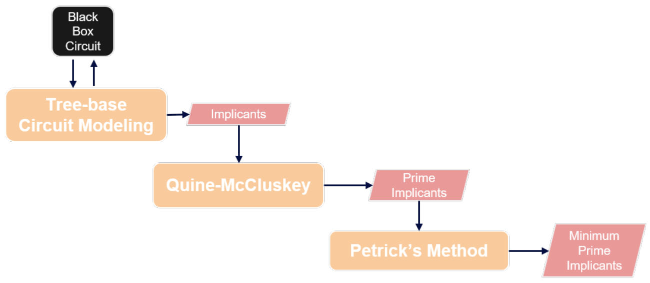


Figure 2: The flow chart of finding the minimum sum of products for a logic function

2.1 Tree-base Circuit Modeling

The Tree-base Circuit Modeling will continuously learn the relationship between input/output from the black box circuit by reading and writing files, and then build implicants. The first step is Ordering input by priority. Then generate the tree by the topX priority input. The next step will test the tree by some input pattern. If there are still have some error input patterns, the Modify tree step will add a new input bit to accurate the tree. While the tree's accuracy fits the requirement, the generate the implicants.

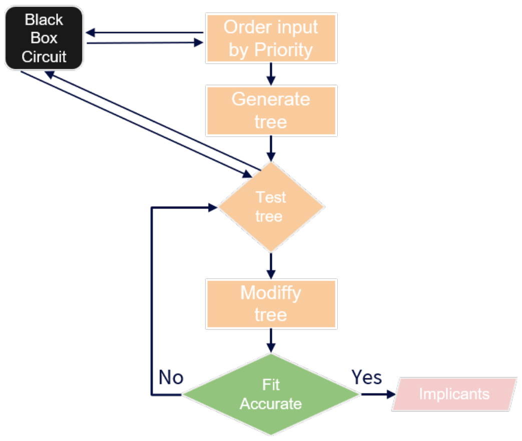
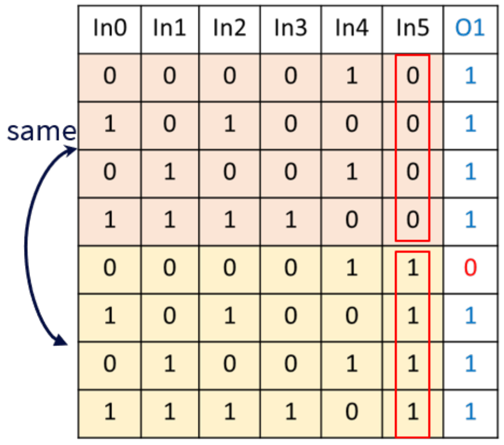


Figure 3: The flow of Tree-base Circuit Modeling

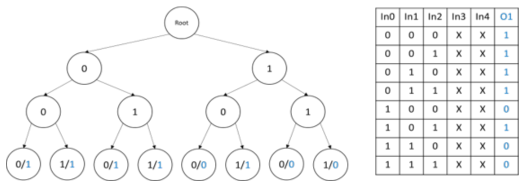
2.1.1 Order input by Priority. The input priority will determine as Fig. 4. We will fix all input patterns except input 5(In5) with all 0/1. The the priority of In5 is the identical ratio of the output with all 0/1. In the Fig. 4 the priority of In5 = 3/4.



In0	In1	In2	In3	In4	In5	O1
0	0	0	0	1	0	1
1	0	1	0	0	0	1
0	1	0	0	1	0	1
1	1	1	1	0	0	1
0	0	0	0	1	1	0
1	0	1	0	0	1	1
0	1	0	0	1	1	1
1	1	1	1	0	1	1

Figure 4: The example of ordering input by priority.

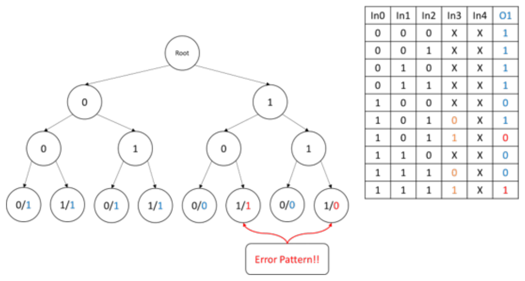
2.1.2 Generate Tree. In this step we will use previous priority of each input to construct the tree. In the Fig. 5, it use top 3 priority input bit to construct the tree.



In0	In1	In2	In3	In4	O1
0	0	0	X	X	1
0	0	1	X	X	1
0	1	0	X	X	1
0	1	1	X	X	1
1	0	0	X	X	0
1	0	1	X	X	1
1	1	0	X	X	0
1	1	1	X	X	0

Figure 5: The example of generating tree.

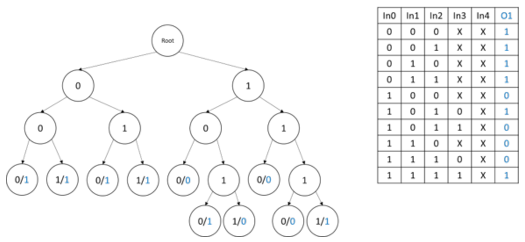
2.1.3 Test Tree. After the tree was generated, we will use some input patterns to test the tree. As shown in Fig. 6, there is two error pattern that occur, which we will fix in the next step.



In0	In1	In2	In3	In4	O1
0	0	0	X	X	1
0	0	1	X	X	1
0	1	0	X	X	1
0	1	1	X	X	1
1	0	0	X	X	0
1	0	1	0	X	1
1	0	1	1	X	0
1	1	0	X	X	0
1	1	1	0	X	0
1	1	1	1	X	1

Figure 6: The example of testing tree.

2.1.4 Modify Tree. When the tree occurs an error, we will use the next priority input bit to fix it. As shown in Fig. 7, we add the 4'th input bit on the error path to fix it.



In0	In1	In2	In3	In4	O1
0	0	0	X	X	1
0	0	1	X	X	1
0	1	0	X	X	1
0	1	1	X	X	1
1	0	0	X	X	0
1	0	1	0	X	1
1	0	1	1	X	0
1	1	0	X	X	0
1	1	1	0	X	0
1	1	1	1	X	1

Figure 7: The example of modifying tree.

2.1.5 Parallelization. We will repeat the Test tree and Modify the tree while the accuracy is fit, then generate the implicants. The implicant in the step has many redundancies, so the following step will minimize the number of implicants. However, we can observe the speed of the program is limitate by "Order input by Priority step" and "Test tree step". These two steps frequent read and write files to get the input and output relation by "Black Box Circuit", so we use OpenMP to parallel read and write files on each circuit input.

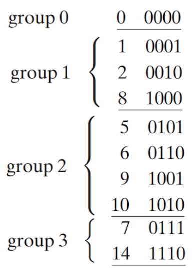
2.2 Quine-McCluskey Method[2][4]

Here we illustrate the algorithm with an example in the textbook, Roth Kinney "Fundamentals of Logic Design". Assume that the logic function which we are intended to find its minimum sum of products is as Fig. 8.

$$f(a, b, c, d) = \Sigma\, m(0, 1, 2, 5, 6, 7, 8, 9, 10, 14)$$

Figure 8

We group the minterms with the same number of 1's as the initial implicant set.

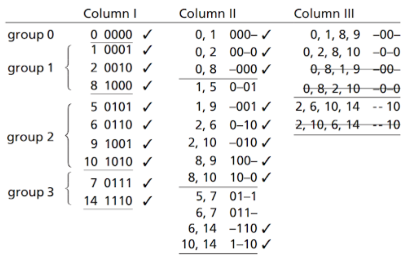


group 0	0	0000
group 1	1	0001
	2	0010
	8	1000
group 2	5	0101
	6	0110
	9	1001
	10	1010
group 3	7	0111
	14	1110

Figure 9

2.2.1 List all the minterms in the initial implicants set. We group the minterms with the same number of 1's as the initial implicant set.

2.2.2 Combine any two of the implicants in the set if possible. Then we check the adjacent groups for the current column to merge the implicants differing in an exact bit and record the new merged implicant in the next column iteratively until we cannot find a pair to merge. We will parallelize the program in this step since the procedure of merging requires the comparison between all the combinations of two implicants in the adjacent groups. The entire algorithm could be speeded up considerably when the dimension of the logic function is high. The implicants without a check mark would be the prime implicants.



	Column I	Column II	Column III
group 0	0 0000 ✓	0, 1 000– ✓	0, 1, 8, 9 –00–
group 1	1 0001 ✓	0, 2 00–0 ✓	0, 2, 8, 10 –0–0
	2 0010 ✓	0, 8 –000 ✓	~~0, 8, 1, 9 –00–~~
	8 1000 ✓	1, 5 0–01	~~0, 8, 2, 10 –0–0~~
group 2	5 0101 ✓	1, 9 –001 ✓	2, 6, 10, 14 --10
	6 0110 ✓	2, 6 0–10 ✓	~~2, 10, 6, 14 --10~~
	9 1001 ✓	2, 10 –010 ✓	
	10 1010 ✓	8, 9 100– ✓	
group 3	7 0111 ✓	8, 10 10–0 ✓	
	14 1110 ✓	5, 7 01–1	
		6, 7 011–	
		6, 14 –110 ✓	
		10, 14 1–10 ✓	

Figure 10: The procedure of combining implicants iteratively

2.2.3 Parallelization. In this section, we discuss about two different implementations of parallelization in OpenMP. By observing the serial version as Algorithm 1, one can easily find that the whole algorithm is about to compare the any two implicants with the difference of 1's count equal to one. We first figure out that utilizing task construct in OpenMp as Algorithm 2 could maximize the parallelism of the whole program. However, due to the computation overhead of the function compareAndMerge() is not expensive, the single thread to trace all over the pairs for comparisons becomes

Algorithm 1 Serial Quine-McCluskey Algorithm

```
    Input: Implicant set $I$ of some boolean function
    Output: Prime implicant set $P$
 1: repeat
 2:     Collect implicants with same number 1's into the same
    group as Fig. 9
 3:     for each adjacent groups $g_1, g_2$ do
 4:         for $i \leftarrow 1$ to $g_1'$s size do
 5:             for $j \leftarrow 1$ to $g_2'$s size do
 6:                 $compareAndMerge(g_1[i], g_2[j])$
 7:             end for
 8:         end for
 9:     end for
10: until Cannot find any implicants pair to merge
```

the main bottleneck of the program and even make the efficiency of the program worse than the serial version at all time. Therefore, we developed another parallelization as Algorithm 2 to achieve better program efficiency compared to the serial version. We simply add a parallel for collapse(2) construct before entering the second for loop to parallelize the work of the comparison between two adjacent groups. Although we could not achieve the maximum utilization rate of threads with this approach when the number of implicant pairs for comparisons in two group is less than the number of thread, we could still get a decent speedup when the input size is large enough as it is shown in section 4.

Algorithm 2 Task-based Parallel Quine-McCluskey Algorithm

```
    Input: Implicant set $I$ of some boolean function
    Output: Prime implicant set $P$
 1: repeat
 2:     Collect implicants with same number 1's into the same
    group as Fig. 9
 3:     #pragma omp parallel
 4:     #pragma omp single
 5:     for each adjacent groups $g_1, g_2$ do
 6:         for $i \leftarrow 1$ to $g_1'$s size do
 7:             for $j \leftarrow 1$ to $g_2'$s size do
 8:                 #pragma omp task
 9:                 $compareAndMerge(g_1[i], g_2[j])$
10:             end for
11:         end for
12:     end for
13: until Cannot find any implicants pair to merge
```

Algorithm 3 For loop-based Parallel Quine-McCluskey Algorithm

```
    Input: Implicant set $I$ of some boolean function
    Output: Prime implicant set $P$
 1: repeat
 2:     Collect implicants with same number 1's into the same group as Fig. 9
 3:     for $each\ adjacent\ groups\ g_1, g_2$ do
 4:         $\#pragma\ omp\ parallel\ for\ collapse(2)$
 5:         for $i \leftarrow 1$ to $g_1's\ size$ do
 6:             for $j \leftarrow 1$ to $g_2's\ size$ do
 7:                 $compareAndMerge(g_1[i], g_2[j])$
 8:             end for
 9:         end for
10:     end for
11: until Cannot find any implicants pair to merge
```

2.3 Petrick's Method[3]

We could get all the prime implicants from a logic function successfully by applying Quine-McCluskey method. Petrick's method is a systematic approach of finding the minimum sum of products of a set of prime implicants. Here, we illustrate this algorithm with another example shown below.

$$F = \Sigma\, m(0, 1, 2, 5, 6, 7)$$

Figure 11

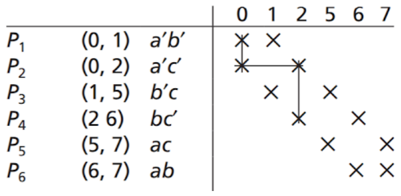
The following are the prime implicants founded by Quine-McCluskey.

0	000	✓	0, 1	00–
1	001	✓	0, 2	0–0
2	010	✓	1, 5	–01
5	101	✓	2, 6	–10
6	110	✓	5, 7	1–1
7	111	✓	6, 7	11–

Figure 12

2.3.1 *Write the logic function P which being true when all the minterms being covered with a set of variables each Pi corresponds to a prime implicant.*

In this section, we would like to treat each prime implicant as a boolean variable first. The prime implicants chart is shown below. The numbers in the most top row are the minterms which should be covered by the set of prime implicants. For each column, we could choose an arbitrary prime implicant with an X mark to cover the minterm. For example, we could choose P1 or P2 to cover the minterm 0 and choose P1 or P3 to cover the minterm 1. So the resulting P function would be written as follows. We could parallelize the procedure of the logic function for each minterm to speed up here and then multiply each function together in the end.



			0	1	2	5	6	7
P_1	(0, 1)	$a'b'$	×	×				
P_2	(0, 2)	$a'c'$	×		×			
P_3	(1, 5)	$b'c$		×		×		
P_4	(2 6)	bc'			×		×	
P_5	(5, 7)	ac				×		×
P_6	(6, 7)	ab					×	×

Figure 13

$$P = (P_1 + P_2)(P_1 + P_3)(P_2 + P_4)(P_3 + P_5)(P_4 + P_6)(P_5 + P_6) = 1$$

Figure 14

2.3.2 *Reduce the logic function P with Boolean equations listed below to sum of products.*

$$\begin{aligned} &1.\ (X+Y)(X+Z) = X + YZ \\ &2.\ X + XY = X \end{aligned} \quad (1)$$

To minimize the boolean equations, it is essential to multiply out the product of sums above. For each multiplication, we could reduce the number of the literals by using the two equations listed above. Again, we could speed up the algorithm by utilizing multiple threads for each multiplication. The procedure of minimization is shown as follows.

$$\begin{aligned} P &= (P_1 + P_2P_3)(P_4 + P_2P_6)(P_5 + P_3P_6) \\ &= (P_1P_4 + P_1P_2P_6 + P_2P_3P_4 + P_2P_3P_6)(P_5 + P_3P_6) \\ &= P_1P_4P_5 + P_1P_2P_5P_6 + P_2P_3P_4P_5 + P_2P_3P_5P_6 + P_1P_3P_4P_6 \\ &\quad + P_1P_2P_3P_6 + P_2P_3P_4P_6 + P_2P_3P_6 \end{aligned}$$

Figure 15

$$P = P_1P_4P_5 + P_1P_2P_5P_6 + P_2P_3P_4P_5 + P_1P_3P_4P_6 + P_2P_3P_6$$

Figure 16

This P function means that all the minterms of the logic function could be covered by the set of prime implicants 1,4,5 or 1,2,5,6 or 2,3,4,5 or 1,3,4,6 or 2,3,6. Thus, there are four combinations of the prime implicants might form the minimum sum of products.

2.3.3 *Parallelization.*

After applying Quine-McCluskey method, we could get a set of prime implicant and construct a resulting P function in product of sums. In this section, we have to multiply out the function P into sum of products, and then find out the product with the minimum cost. We implemented two methods to solve the parallelization problem.

2.3.4 *Method 1.*

For the first method, we applied multitasking to parallel the process. A single thread was assigned to multiply out the function P, which would return a product after performing a multiplication. Other threads were assigned to calculate the cost of returning products. However, this method didn't achieve a good parallelization effect. By reducing the number of threads to one, we found that the runtime spent on multiplying the function P is much greater than the runtime spent calculating the costs. In other words, except for the single thread that is assigned to multiply out the function P, other threads have a lot of time waiting for the single thread. Therefore, we figured out the second method to solve the parallelization problem.

2.3.5 Method 2. In the second method, we separate Petrick's Method into two parts, and apply "parallel for" on both parts. First, all threads multiplied out the function P and stored the resulting products into the memory. After that, all threads calculated the costs of the resulting products stored in the memory. Applying the second method, we got a good parallelization effect. But this method still remain a disadvantage, it requires a huge memory storage space to store the products generated during the process.

3 EXPERIMENTAL METHODOLOGY

Our Experimental is run on the Intel(R) Xeon(R) Silver 4208 CPU @ 2.10GHz * 2 with 16 cores, 32 threads and 64GB RAM. The input data is from the ICCAD-2019 CAD contest. We use different numbers of threads to test our program in each step, then show the experimental result.

4 EXPERIMENTAL RESULTS

In our experiments, the speedup converged to 4.5 when we increasing the thread number which follows the Amdahl's law. Also, one could easily find the fact that the speedup of overall program and tree-based circuit modeling is almost the same. In other words, the overall program runtime is dominated by tree-based circuit modeling which implies the computation overhead for logic minimization of input cases of our experiments is not high.

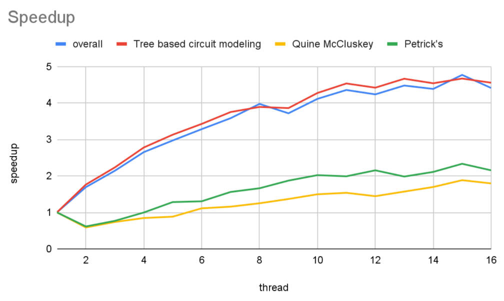


Figure 17: The speedup of different thread number

As for the speedup for Quine-McCluskey and Pertrick's method, our implementations are not really effective but still better than the serial version when the thread number is higher than 4. The overhead of parallelism is most obvious when the thread number is 2, which leads to an even worse program performance compared with the serial version. Since the algorithm of each requires synchronizations among threads frequently during the program procedure which is in fact inevitable, the speedup is limited by the algorithm intrinsically. Nevertheless, we believe that our implementations are still efficient when the size of the input boolean function is as large as the real circuit design with thread number higher than 4.

5 RELATED WORK

The first place of the 2019 ICCAD Contest proposed a decision-tree based circuit learning algorithm and template based preprocessing [1]. First, use template based preprocessing to recognize whether the behavior of the circuit is a Comparator or Linear arithmetic. Second, use a decision-tree based circuit learning algorithm with an exhaustive sampling up to 2^{18}. Restricted by the rules of the competition, they can't use parallel programming to speed up the program. So, we want to explore the potential of using the parallel programming skill on the logic synthesis problem.

6 CONCLUSIONS

In this paper, we provide a complete parallelized program of logic regression on high dimensional Boolean space, including the implementation of tree-base circuit modeling, Quine-McCluskey method and Petrick's method. We discussed the reasons for the difficulties encountered during parallelization and figured out better ways to solve them. Experimental results show that the speedup and the number of threads are nearly positively correlated.

REFERENCES

[1] Pei-Wei Chen, Yu-Ching Huang, Cheng-Lin Lee, and Jie-Hong Roland Jiang. 2020. Circuit learning for logic regression on high dimensional boolean space. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.
[2] Edward J McCluskey. 1956. Minimization of Boolean functions. *The Bell System Technical Journal* 35, 6 (1956), 1417–1444.
[3] Stanley R Petrick. 1956. A direct determination of the irredundant forms of a Boolean function from the set of prime implicants. *Air Force Cambridge Res. Center Tech. Report* (1956), 56–110.
[4] Willard V Quine. 1952. The problem of simplifying truth functions. *The American mathematical monthly* 59, 8 (1952), 521–531.
[5] Ulf Schlichtmann, Sabya Das, Chao Lin, and Mark Po-Hung Lin. 2019. Overview of 2019 CAD Contest at ICCAD. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–2.