COMS 4995 Parallel Functional Programming Project Report

Jian Song (js5316), Ziao Wang (zw2498)

December 16, 2019

1 Introduction

MapReduce is a programming model and an associated implementation for processing and generating large data sets[1]. Some large tasks with specific pattern can be easily split into map parts and reduce parts to be implemented in parallel and sped up. Typically, a map function processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function merges all intermediate values associated with the same intermediate key.

In our final project, we solved a typical problem of Subset Sum in parallel using MapReduce-style algorithm. We also tried to solve a similar permutation problem of N-Queens. We tested our code on a virtual machine of ubuntu with 8 cores in serial and parallel separately and evaluated the performance and speedup of our program.

Please follow the instructions in README.md for compiling and running the program.

2 Subset Sum Problem

Subset problem is a combination problem. The problem itself is easy to understand: given a set of numbers and a target value k, find how many different combinations of elements in the set sum up to k. This has been proved as a NP-hard problem, which means we are unable to solve it in polynomial time.

2.1 Serial Algorithm

Normally, the algorithm is a recursive-style one. At each step, we have two choices, that is whether add the current element or not, and then we keep scanning the rest of the set. We can implement this in less than 5 lines in Haskell.

Since that equals to calculate the sum of all subsets, the time complexity would be $O(2^n)$, where n stands for the size of the set.

2.2 Parallel Algorithm

As can be seen in serial algorithm, the exponential function could be significantly slow when n increases, and it might cause stack overflow as well. Therefore, we are here to suggest a MapReduce-style parallel algorithm to solve this problem.

2.2.1 Map

Since we have to consider each subset, there is no way to get rid of calculating the sum of 2^n subsets. However, we could use multiple cores to do that in parallel. We firstly encoded those 2^n subsets in binary format. E.g. We have a set of 2 elements {1,2}, then we will have four subsets encoded as 00, 01, 10, 11. Here 1 means that the element is picked and 1 is not. Each core would take care of a chunk of numbers range from $(((2^n)/N_m) * (i-1) + 1$ to $((2^n)/N_m) * i$, where N_m stands for the number of cores and n stands for the size of the set. After calculating the sum, the map function will compare the sum with target value, and will return 1 or 0 as the output.

2.2.2 Reduce

The reduce function will sum all values from Map. Since we only want a MapReduce-style program rather than a real MapReduce project, we didn't use log files as the intermediate output because that would involve unnecessary IOs as well as communications between processes/threads.

2.2.3 Pseudo Code

Algorithm 1

```
input set S, target K, size n
for x in (((2^n)/N_m) * (i − 1) + 1 to ((2^n)/N_m) * i do
    b = BinaryFormat of x
    curSum = 0
    for i in length(b) do
        if b[i] == 0 then
            curSum += S[i]
        end if
    end for
    if curSum == K then
        output 1
    else
        output 0
    end if
end for
return sum of all outputs
```

Table 1. Performance of Subset Sum on 1-8 Cores

Cores	Time(s)	Speedup
1	68	1.00
2	37	1.84
4	22	3.09
8	16	4.25

2.3 Evaluation of Algorithm

We evaluated the program in the following metrics[1]:

- Fix the size of set and run program with different number of cores.
- Fix the number of cores and run program with different size of set.

2.3.1 Set with 25 Elements

For a set with 25 elements, there are $2^{25} = 33554432$ subsets. We tested the program separately using 1,2,4,8 cores. The result is as shown in Figure 1 and Figure 2.

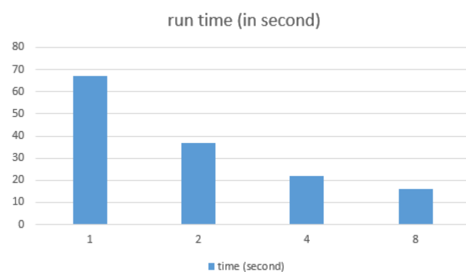

Figure 1. Run Time

Figure 2. Balance

We can say that the program is indeed paralyzed and it is 'linear' since the run time reduced by almost half as the number of cores increase twice. One thing to notice is that we only used a serial reduce worker to do the reduce, therefore this implies the time is dominated by map process. As the result shown in Table 1, we achieved a total speedup of 4.25 on 8 cores comparing to running in serial.

2.3.2 Run Program with 8 Cores

We tested $n = 20$ to 25 with 8 cores, and the result is shown as in Figure 3.

[1]Data is randomly generated via a script.

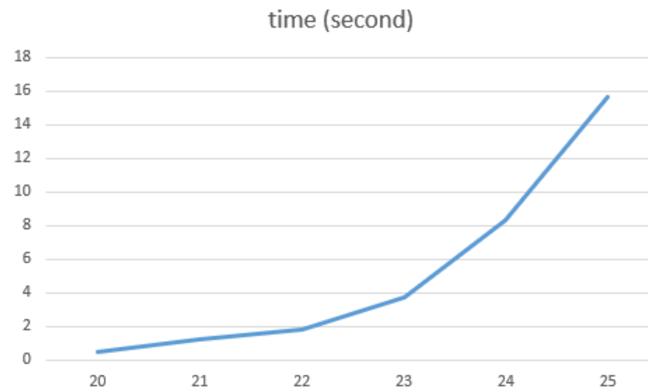

Figure 3. Run Program under Different Set Sizes

As can be seen in the figure, the time becomes exponentially as we expected a NP-hard problem will be.

3 N-Queens Problem

As permutation problems have similar patterns with combination problems, after solving the Subset Sum problem in parallel, we also wanted to try to solve a permutation problem of N-Queens in parallel. The N-Queens problem is the problem of placing n queens on an n*n chessboard such that no two queens attack each other. The number of distinct solution will be returned. The rule of queens' attack is that a queen can attack any other queen which is on the same row or column or diagonal with itself.

3.1 Serial Algorithm

Aiming to maximize the speedup, we implemented a recursive-style brute-force algorithm in serial at the beginning. We first generate all the possible permutations of the n queens position. Each permutation is represented as a list with elements ranging from 1 to n. In this way, each permutation actually enforces that any two queens in this permutation won't be on the same row (represented by list index) and also won't be on the same column (represented by list element.) Then we filter the permutations which have two queens on the same diagonal and finally we get all the valid solutions.

The time complexity of this brutal algorithm is $O(n!)$ as we need to generate all the permutations of a list ranging from 1 to n, where n stands for the length of the board and the number of the queens. The factorial function could be significantly slow when n increases, and it might also cause stack overflow as well.

3.2 Parallel Algorithm

We tried to improve and parallelize the program based on the serial algorithm. There are mainly two parts in the program that can be very time-consuming in a factorial way. First, the generation of all the permutations is in factorial of n; Second, the filtering part to get

the valid solutions which don't have any diagonally aligned queens, which is also in factorial of n.

The second part can be easily parallelized. We implemented our own *parFilter* at first to check each solution valid or not in parallel. But this approach, as expected, leaded to overhead as there are too many permutations and not enough work for each permutation. The overhead was dominated and sparks were overflowed. So we decided to split works into larger chunks using *parList*. By tuning an appropriate number of chunks to increase each part's workload, we made all the sparks generated are successfully converted.

The first part is a little tricky to be parallelized. Although it is recursive-style, there is no stack in the bottom that can be used in common. Assume we want to parallelize a permutation of n, we can parallelize the first element of the list and then let each thread add the tail of the list. In this approach, each thread will have to do the permutation of $n-1$, which is also $O((n-1)!)$. As the impact of parallelism to speedup a permutation problem from $O(n!)$ to $O((n-1)!)$ is pretty trivial when the n increases, we decided not to parallelize the first part and implemented it with the built-in *permutations* method from Data package.

3.3 Evaluation

We evaluated the parallel program with a fixed chunk size of 32 (empirical value) with different number of cores. For a given n of 10, there are 10! = 3628800 permutations. We tested the program separately using 1,2,4,8 cores. The time performance is pretty much the same and not quite improved by running in parallel with multiple cores. We looked into the eventlog and found it is bounded by the first part of the generation of the permutation. Although we can parallelize the second part of filtering from $O(n!)$ to $O(n!)/m$, where n stands for the size of the board and m stands for the number of the cores we are using, the first part in the program remains as $O(n!)$. The second part is only executed after we are done with the first part and get all the permutations. So the overall time performance for the program remains as $O(n!)$ which is basically the same with running in serial.

4 Future Works

For Subset Sum, we did the reduce part in serial, and this could possibly be paralyzed by using other data types like Repa array. Besides, the program will cause segment fault when the size exceeds 29.

5 Conclusion

In this project, we parallelized two NP problems of Subset Sum and N-Queens in Haskell. We effectively parallelized Subset Sum and achieved a speedup of 4.25 using 8 cores for the combination problem. For N-Queens, the program is parallelized but the performance is still bounded by the permutation part which can't quite be effectively parallelized.

6 Appendix

Listing of the Code

- subsetsum.hs
- nQueens.hs
- READMe.md: compiling and running instructions

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.