Report for COMP90025 Project 2

Zichun Zhu 784145 zichunz, Yijian Zhang 806676 yijianz

1 Introduction

The knapsack problem comes up fairly often in parallel computing, this is an interesting problem as it is easy to solve sequentially, however, kind of tricky to solve in parallel. In this report, we give a brief description of knapsack problem at first, we start with an sequential algorithm of knapsack problem, then presents parallel methods for dealing with the task, implemented by MPI and OpenMP respectively. After some analysis of related results, a brief conclusion for parallel knapsack problem is given.

2 Background

The knapsack problem is an optimization problem derives from a scenario where one is constrained in the number of items that can be placed inside a fixed-size knapsack. Given a set of items with specific weights and values, the aim for the problem is to get as much value into the knapsack as possible given the weight limitation of the knapsack. There are mainly two types of the knapsack problem, which are 0-1 knapsack problem and bounded knapsack problem. For 0-1 knapsack problem, it restricts the number of copies of each kind of item to zero or one, while for bounded knapsack problem, this restriction has been removed. This report focus on the 0-1 knapsack problem.

2.1 The 0-1 Knapsack Problem

Assume that we have a knapsack with max weight capacity W=5, our objective is to fill the knapsack with items such that the value of items is maximum. For example, following table contains the items along with their values and weights.

Item	1	2	3	4
Value	100	20	60	40
Weight	3	2	4	1

Table 1: example for the 0-1 knapsack problem

Then we create a value table V[i,w], where i means number of items and w means the weight of the items.

V[i,w]	W=0	1	2	3	4	5
i=0						
1						
2						
3						
4						

Table 2: Value table

The rule to fill the V[i,w] table can be described by the following pseudo code.

```
**if** wt[i] > w **then**
    V[i,w] = V[i-1, w]
**else if** wt[i] <= w **then**
    V[i,w] = max(V[i-1, w], val[i] + V[i-1, w-wt[i]])
```

Figure 1: Pseudo code for filling value table

After calculation the value table is filled as below.

V[i,w]	W=0	1	2	3	4	5
i=0	0	0	0	0	0	0
1	0	0	0	0	100	100
2	0	0	20	100	100	120
3	0	0	20	100	100	120
4	0	40	40	100	140	140

Table 3: Finished value table

Hence the maximum value earned is V[4,5] = 140, so item we choose are 4 and 1. The time complexity of the 0-1 knapsack problem is O(nW), where n is the number of items and W is the capacity of knapsack.

3 Methodology

3.1 Sequential Algorithm

Following pseudo code is the dynamic programming for solving the 0-1 knapsack problem.

```
for j from 0 to W do:
    m[0, j] = 0
for i from 1 to n do:
    for j from 0 to W do:
        if w[i] > j then:
            m[i, j] = m[i-1, j]
        else:
            m[i, j] = max (m[i-j, j], m[i-1, j-w[i]] + v[i])
```

Figure 2: Pseudo code for sequential solution

The first loop is to fill the first row with 0. The nested for-loops is to parse the matrix with a row, and at each iteration, the maximum value is chosen between the value with new item(m[i-1, j-w[i]] + v[i]) and the value without using new item(m[i-j, j]).

3.2 OpenMP Implementation

The 0-1 knapsack problem can be solved using data parallelism as the same computation is done on every cell of the data. According to the sequential algorithm, we noticed that the value of a cell can either be m[i-1, j-w[i]] + v[i] or m[i-j, j], so there are dependency between the rows(item) rather than the columns(weight), which means for each row the column is performed simultaneously. Thus the problem can be solved by column parallelization.

In OpenMP optimization, each column of the matrix is mapped to a processor. Comparing to the sequential algorithm, the difference is that only an array of two rows is used, because it is only necessary for getting two elements from the previous row when computing every element. Hence generating a new instance for the knapsack problem at first, once all the weights are calculated column wise, compare all the weights and find the best one.

3.3 MPI Implementation

In MPI optimization, each column of the matrix is mapped to a processor. At each iteration the execution contains some steps, firstly compute the maximum value using the new item, the rank use MPI_Recv to get the value of the [i-1, j-w[i]] + v[i] from the rank has compute this value. Then compute the value without the new item, save the chosen item with its value, finally use MPI_Isend to send to all the processors the new value for future need.

4 Result

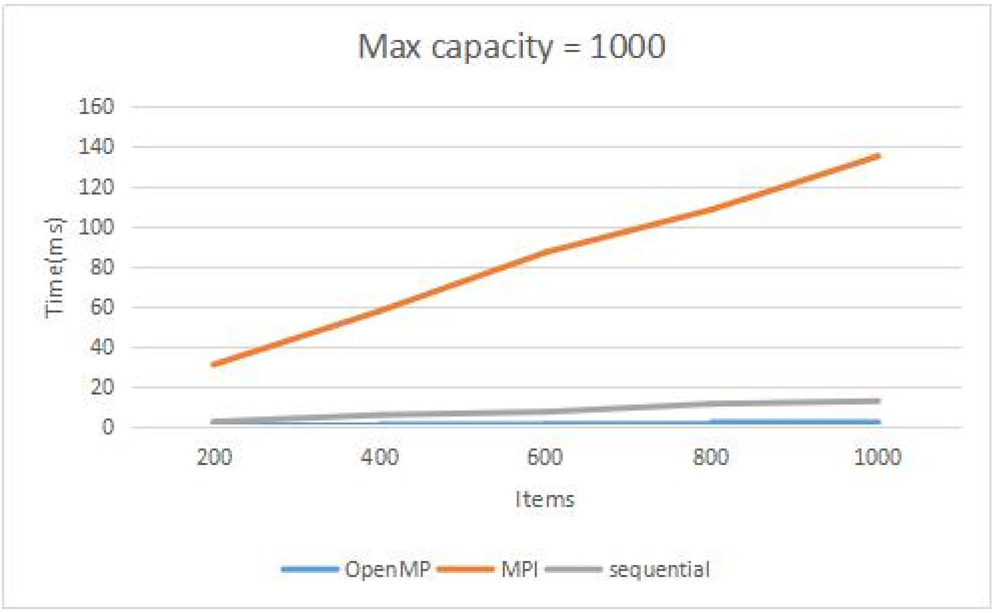

Figure 3: Result for the 0-1 knapsack problem

According to the chart above, the OpenMP program performs better than sequential program, which verify that the column can be solved independently in this problem, hence it is efficient for applying parallization for each column. The parallel methods based on the MPI program performs the worst comparing to other methods. One of the potential reasons is that the matrix is still too small for the MPI to perform better, when the matrix is large enough it might better than sequential execution. Furthermore, the messages passing might be not efficient, plenty of messages are sent leading unexpected delay for find the best value during MPI_Recv process.

5 Conclusion

In general, parallization for 0-1 knapsack problem is feasible with respect to our experiment. OpenMP provided satisfied result that show our parallel methods is useful for dealing with the task. As there are some unexpected results, which lead us to think about some improvements. First of all, larger data set could be taken into account to evaluate the efficiency of the programs, then MPI program should be optimized or figure out the where the delays appears, in addition, more solution such as CPU based using CUDA could be used to compare efficiency of solving the 0-1 knapsack problem.