COMP90025 Parallel and Multicore Computing
Project 1B Report

Hanbin Li *hanbinl1*
Wenqing Xue *wenqingx*

September 2019

Introduction

The goal of this project is to solve the knapsack problem by implementing the MPI[1] program. There are two sequential algorithms normally implemented, brute-force and DP[2] approaches. In detail, the complexities of brute-force and DP are $O(2^n)$ and $O(nC)$ correspondingly. By contrast, only tiny values of n can make brute-force approach more comparable. Thus, we use DP approach as our base methodology for parallelism.

Dependency Analysis

For sequential DP algorithm, it uses a 2-dimension matrix to represent the computations' evolution (row as items, column as weights) and each cell contains the best value to include without exceeding the weight of its column. To do so, a row-oriented 2 for-loops structure is implemented to parse the matrix. At each iteration, it first checks if the current item's weight is higher than the column weight, if so, the maximum value is the value of the row above. Otherwise, it computes the maximum value between the value for the same weight without the new item (cell[i-1][j]) and the value using this item plus the best value for this column weight (cell[i-1][j-weight]+v[i]). At the end of computation, the best value could be found in the last cell, which is cell[item-1][max weight].

By analysing the sequential algorithm, we realised that there is no task parallelisation realisable as all the task depends on each other. However as every cell of the matrix is performing the same computation, we could parallelise the algorithm using data parallelism. The above algorithm shows that the value of the cell is either cell[i-1][j] or cell[i-1][j-weight]+value[i]. Thus, we can see the dependencies are only between the rows but not between columns. The cell only needs to know the value of the row above itself and the value of the previous column is independent. Therefore, we decided to implement column-level parallelism to allow all the column to perform simultaneously.

[1]MPI: Message Passing Interface
[2]DP: Dynamic Programming

MPI Approach

In order to achieve parallel approach, pseudo code of MPI program with dynamic programming is shown below.

Algorithm 1 MPI Parallel Algorithm

```
for $i \leftarrow 0$ to $n-1$ do
    for each node $j$ $(rank \leq j \leq C, j+=size)$ do
        if $j < w[i]$ then
            $cost = 0$
        else if $i == 0$ and $j \geq w[i]$ then
            $cost = v[i]$
        else
            $cost = K[i-1][j] + v[i]$
        end if
        if $i == 0$ then
            $K[i][j] = max(cost, 0)$
        else
            $K[i][j] = max(cost, K[i-1][j])$
        end if
    end for
end for
```

Column parallelisation is applied to our MPI implementation. Each columns of the 2D matrix is mapped to a processor. The mapping is made to divide the work evenly so that each process could work on the column. At each row, each processor computes the column it is mapped to and the execution contains following steps:

1. For the item of the row, compute the maximum value with the column weight.
    (a) If the column's weight is smaller than the weight of the item, the value is 0.
    (b) If it is the first item and the column's weight is bigger than the weight of the item, the value is the item's value.

   (c) If it is not the first item and the column's weight is bigger than the weight of the item, the value is the item value plus the value of cell[i-1][j-weight[i]] (received from other process).

2. Compare the value with this item and the value without this new item, which is the value in the above row.
3. Record the maximum value into the cost table.
4. Check if current column weight plus the weight of next item is less than the capacity, if so, send this value to process that could need it in future iteration.

Discussion

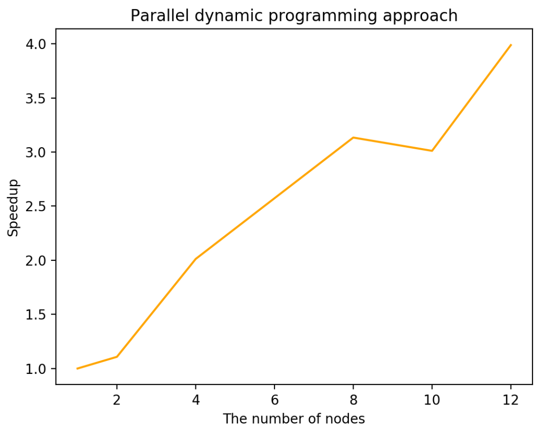

Figure 1: Speedup comparison with nodes

As shown in the figure above 1, the performance of MPI parallel approach is proportional to the number of nodes, although the speedup scale is around one third of nodes number. There are several reasons which may cause this phenomenon, for example, interdependencies between computations which makes the cost of communications increase significantly. Additionally, for different nodes, it takes different time to complete the algorithm, therefore the management complexity may go up.

n*C	Parallel(us)	DP with OpenMP (us)
100000	64715	2856
250000	157551	4588
500000	285675	4327
750000	467453	7049
1000000	578182	6750
2500000	1437215	13686
5000000	3295152	26006

By comparing the runtime of two methods, we found that DP with OpenMP approach spends significantly less time than MPI parallel approach, which is the main reason we use this as our submission. The reason behind that may because that the message passing is not quite efficient, since lots of messages are sent at each iteration loop. Even though we use MPI_Isend asynchronous operation for non-blocking sending, there are still some blocked waiting time for data to be available in the buffer in receive operations, which causes the unexpected delay.