# Analysis for DP Approach

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Items** | **Capacity** | **Sequential** | **2 Threads** | **4 Threads** | **8 Threads** | **16 Threads** |
| **100** | **100** | **0** | **0** | **0** | **0** | **0** |
| **100** | **10000** | **15** | **38** | **15** | **13** | **17** |
| **100** | **1000000** | **2544** | **1391** | **1064** | **627** | **551** |
| **10000** | **100** | **16** | **219** | **246** | **320** | **534** |
| **1000000** | **100** | **1914** | **17742** | **25151** | **33794** | **53196** |

Improvements in parallel code:

1. While we fill up the table in DP approach, we need values only from previous rows. Therefore, instead of having table of size of number of items we can have a table of two rows.
2. I used **#pragma omp parallel for** construct to parallelize the for loop as values in the same row doesn’t depend on each other. As the parallelization is done over a row, the time taken is definitely affected by number of columns i.e Knapsack capacity
3. There’s no improvement over number of items in sequential and parallel code. All the items need to be traversed sequentially in both the approaches.

**Complexity:**

**Sequential**

We need to create a table of size n\*C and fill each cell by traversing.

n: Number of items and

C: Knapsack Capacity

TimeComplexity: O(n\*C)

**Parallel**

If we have t threads, each thread will be assigned W/t columns. Therefore, for loop will get executed in O(W/t) unlike O(W) for serial one.

Now we need to iterate through all the items and execute this for loop everytime.

Overall time complexity: O(n\*(W/t))