Knapsack Algorithm

Presentation by Shrishti Karkera (50485408)
Overview

Problem Definition
- Sequential approach
- Sequential implementation

Observation
- Parallel approach
- Parallel implementation

Results

Observation
0/1 Knapsack

W <= Total weight
Max Total value
Recursion

def knapsack(W, wt, val):

    Base Case
    if n == 0 or W == 0:
        return 0

    not_pick = knapsack(W, wt, val, n-1)

    Conditions
    pick = -1e9
    if (wt[n-1] <= W):
        pick = val[n-1] + knapsack(W-wt[n-1], wt, val, n-1)
    return max(pick, not_pick)
Recursion with memoization

**Base Case**

```
def knapsack(W, wt, val):
    if n == 0 or W == 0:
        dp[n][W] = 0
        return 0
    not_pick = knapsack(W, wt, val, n-1)
```

**Conditions**

```
pick = -1e9
if (wt[n-1] <= W):
    pick = val[n-1] + knapsack(W-wt[n-1], wt, val, n-1)
```

```
dp[n][W] = max(pick, not_pick)
```

**dp = 2d array (n+1 x W+1)**

```
dp[i][w] = dp(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w])
```
weights = [3, 4, 7]
values = [4, 5, 8]
W = 7

Tabular DP

\[
dp[i][w] = dp(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w])
\]

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>9</td>
</tr>
</tbody>
</table>

Max profit
Approach 1 - 1 column per core

values = [4, 5, 8]

```
value = [4, 5, 8]
weight = [3, 4, 7]
W = 7

memory = {}
for i in range(rows):
    memory.append([0]*cols)
start_time = MPI.Wtime()

# For each column --> through rows
for i in range(1, rows):
    # send data
    if rank < size - weight[i-1]:
        comm.send(memory[i-1][0], dest = rank + weight[i-1])
    # receive data
    if rank >= weight[i-1]:
        fetchedValue = comm.recv(source = rank - weight[i-1])
    # compute
    if weight[i-1] > rank:
        memory[i][0] = memory[i-1][0]
    else:
        memory[i][0] = max(memory[i-1][0] + fetchedValue, memory[i-1][0])
```
Approach 2 - multiple columns per core

values = [4, 5, 8]

Iterate
1. Send data
2. Receive data
3. Calculate for the current cell

Code

```python
if min_col_per_node * size - 1 >= W:
    cols = min_col_per_node
else:
    result = W + 1
    while result % size != 0:
        result += 1
    cols = result // size
memory = []
for i in range(rows):
    memory.append([0] * cols)
start_time = MPI.Wtime()

# Initialize 0th Row
for j in range(cols):
    memory[0][j] = j + (cols * rank)

# Initialize Remaining Rows with zero value
for i in range(1, rows):
    for j in range(cols):
        memory[i][j] = 0
```
Standard execution

Amdahl’s Law

\[ S_p \leq \frac{1}{(1 - f) + \frac{f}{p}} \]

\( f \) is the fraction of the program that must be executed serially (i.e., cannot be parallelized) and \( p \) is the number of processors.
Gustafson’s Law

\[ S_p = p - (p - 1) * f \]

where

\( f \) is the fraction of the program that is inherently serial and \( p \) is the number of processors.
Speed-up

\[
\text{Speed-up} = \frac{T_{\text{sequential}}}{T_{\text{parallel}}}
\]
References

https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/
https://www.educative.io/answers/difference-between-amdahls-and-gustafsons-laws
https://www.stolaf.edu/people/rab/pdc/text/alg.htm#:~:text=to%20be%20avoided.-,Speedup,we%20have%20n%2Dfold%20speedup.
Thank you!

Feel free to ask questions