ITCS 6145

Parallel Computing
Professor : Jinzhen Wang

Name : Aravinda Reddy Gangalakunta
Student Id : 801393674

Assignment 1 : Extracting Parallelism

---

1) Coin Collection

Several coins are placed on an n×m board, with at most one coin per cell. A robot starts at the upper-left corner of the board and can only move either **right** or **down**. The robot collects a coin whenever it visits a cell with one. The objective is to determine the maximum number of coins the robot can collect when it reaches the bottom-right corner of the board. The robot's movement is constrained: it cannot move **up** or **left**.

```
 int collectCoins(int rows, int cols, int[][] coinBoard) {
    int[][] maxCoins = new int[rows][cols]; // No.of coins can be collected
while on (i, j)
    // Initialize the first cell
    maxCoins[0][0] = coinBoard[0][0];
    // Fill the first row
    for (int col = 1; col < cols; ++col) {
        maxCoins[0][col] = maxCoins[0][col - 1] + coinBoard[0][col];
    }
    // Fill the first column
    for (int row = 1; row < rows; ++row) {
        maxCoins[row][0] = maxCoins[row - 1][0] + coinBoard[row][0];
    }
    // Fill the rest of the table
    for (int row = 1; row < rows; ++row) {
        for (int col = 1; col < cols; ++col) {
            maxCoins[row][col] = Math.max(maxCoins[row - 1][col],
maxCoins[row][col - 1]) + coinBoard[row][col];
        }
    }

    return maxCoins[rows - 1][cols - 1];}
```

1) **Complexity :**

The Overall time complexity of this function is O(rows*cols).

2) **Extract dependencies :**

Each cell maxCoins[row][col] depends on:

- maxCoins[row-1][col] (the cell above)
- maxCoins[row][col-1] (the cell to the left)

The first row (row = 0) cells depend only on the cell to their left and the input.
The first column (col = 0) cells depend only on the cell above them and the input.

All other cells depend on the maximum of the cell above and the cell to the left, plus the input value. (`Math.max(maxCoins[row - 1][col], maxCoins[row][col - 1]) + coinBoard[row][col];`)

3) **Width :** Refers to the maximum number of computations that can be performed in parallel at any given time.

```
// rows (cannot be parallelized due to dependency on previous column)
for (int col = 1; col < cols; ++col)
{
maxCoins[0][col] = maxCoins[0][col - 1] + coinBoard[0][col];
}

// columns (cannot be parallelized due to dependency on previous row)
for (int row = 1; row < rows; ++row)
{
maxCoins[row][0] = maxCoins[row - 1][0] + coinBoard[row][0];
}
```

```
// Partially parallelizable: Each Anti diagonal can be computed in parallel
for (int row = 1; row < rows; ++row) {
    for (int col = 1; col < cols; ++col) {
        maxCoins[row][col] = Math.max(maxCoins[row - 1][col],
maxCoins[row][col - 1]) + coinBoard[row][col];
    }
}
```

When computing maxCoins[1][1], maxCoins[1][2], and maxCoins[2][1], they all depend on maxCoins[0][0], maxCoins[0][1], and maxCoins[1][0]. So, the cells on the next diagonal can be calculated in parallel.

So to summarize Width : min(rows,cols)

4) **Work:** In the function above, every cell in the `maxCoins` table is updated exactly once, and each update involves a constant amount of work (a comparison and an addition). So total work will be just rows multiplied by columns

Work : O (rows * cols)

5) **Critical Path :** The longest sequence of dependent computations from start to finish.

In above function it has to travel from 0,0 to rows-1, cols-1 which means

Critical Path Length: rows + cols

2) **Knapsack**

In the *knapsack problem*, you need to pack a set of items, with given values and sizes (such as weights or volumes), into a container with a maximum capacity . If the total size of the items exceeds the capacity, you can't pack them all. In that case, the problem is to choose a subset of the items of maximum total value that will fit in the container.

```
void knapsack(int numItems, int maxWeight, int[] itemValues, int[]
itemWeights, int[][] dpTable) {
    // Initialize the first row for the base case when no items are
considered
    for (int weight = 0; weight <= maxWeight; ++weight) {
        dpTable[0][weight] = 0;
    }

    // Iterate over all items
    for (int item = 1; item <= numItems; ++item) {
        // Iterate over all weights from 0 to maxWeight
        for (int weight = 0; weight <= maxWeight; ++weight) {
            // Initialize with the value if the item is not included
            dpTable[item][weight] = dpTable[item - 1][weight];

            // Check if the current item can fit in the knapsack
            if (itemWeights[item - 1] <= weight) {
                // Update the value to the maximum between including or not
including the item
                dpTable[item][weight] = Math.max(dpTable[item - 1][weight],
                                                 itemValues[item - 1] +
```

```
dpTable[item - 1][weight - itemWeights[item - 1]]);
            }
        }
    }
}
```

1) **What is the complexity of function ?**

The Overall Time complexity of the function is O(n*W) i.e. (number of Items * weight+1) as it contains two nested loops.
Similarly space complexity for dpTable is (numItems + 1) * (maxWeight + 1)

2) **Extract Dependencies**

For every value of dpTable[item][weight] depends on the value for each item at capacity, you need to know the value for the previous item at the same capacity and the capacity reduced by the item's weight.

Each cell dpTable[item][weight] depends on:

- dpTable[item-1][weight] (always) (Above cell)
- dpTable[item-1][weight - itemWeights[item-1]] (when itemWeights[item-1] <= weight)
- A diagonal pattern in the dpTable, where each cell potentially depends on the cell directly above it and another cell to its left in the row above.

3) **Width** : The number of independent computations that can be done in parallel

.

This Part of code can be parallelized because its independent computations and just initialization

```
for (int weight = 0; weight <= maxWeight; ++weight) {
        dpTable[0][weight] = 0;
    }
```

```
for (int item = 1; item <= numItems; ++item) {
    //Inner loop can be parallelized
   for (int weight = 0; weight <= maxWeight; ++weight) {
        dpTable[item][weight] = dpTable[item - 1][weight];
        if (itemWeights[item - 1] <= weight) {
            dpTable[item][weight] = Math.max(
                dpTable[item - 1][weight],
```

```
                itemValues[item - 1] + dpTable[item - 1][weight -
itemWeights[item - 1]]
            );
        }
    }}
```

Inner loop can be parallelized row by row i.e. only after completing one row we can be able to parallelize complete set of next items i.e. within in same iteration it's only depending on previous row not on current row values

Dependencies **between iterations** of the inner loop (i.e., `weight` iterations) prevent parallelization.

Dependencies **on prior, fully computed data** (i.e., previous `item`) do not prevent parallelization within the current loop.

So in summary total width is not greater than **O(max weight or total capacity)**

4) **Work :** Total amount of computation performed (sum of all operations across all iterations).

   So this is straightforward: it's represented by all operations in the nested loops.

   The total work of the algorithm is O(n * W), where n is the number of items and W is the knapsack capacity.

5) **Critical Path :** The sequence of dependent operations that forms the longest path, meaning these must be executed sequentially.

The outer loop item has a dependency chain, where each iteration depends on the previous one.

- The inner loop weight also has a dependency chain, but it's shorter than the outer loop's chain.
- The critical path follows the outer loop's dependency chain, as it's the longest.

The length of the critical path is equal to the number of iterations in the outer loop:

Critical Path Length = **numItems**