Parallel Dynamic Programming

ECE 361C Term Project
Group 11: Cole Berkley, Zihao Hong, Jiaju Wang, Aarya Atluri

Introduction:

Dynamic programming is a method for solving complex problems by breaking them into different subproblems, finding the optimal solution for each of them, and combining all optimal solutions to get the final result. LLP(lattice-linear predicate) is a powerful approach to formalizing parallel methods in solving sequential problems by constantly updating solution vectors given the forbidden predicates. In this project, we parallelly implemented four dynamic programming problems using the LLP approach and compared their runtime with their sequential implementations.

Longest Increasing Subsequence

To find the longest increasing subsequence, we created an array which stores the length of the longest increasing subsequence at each index i, and comparing indexes j that are less than i and incrementing the max length of the sequence based on an initial max length of 1 for the current element and taking the max of the array returns the longest increasing subsequence.

During my implementation I first set all G's to zero initial, as each G go through advance, their fixed value turns true, in the forbidden, each thread detects if there exist an pre(smaller than G[j]) that's not fixed, if all pre are fixed precede to advance, in advance we make make G[j] the increment of the max pre and set fixed[j] to true,

One problem I encountered is that I wrote G[j]=G[i].incrementandget() at first which ended up incrementing the G[i] too which is not something we want, I then changed it to G[j].set(G[i].get+1) and this fixed the problem.

The parallel example, {35,38,27,45,32}, I get the following table for my implementation (1)

Parallel Runtime complexity for this is going to be O(klogN) where k is the length of the longest sequence, N is the length of the input array since each thread only checks for those to the left of G[j].

Optimal Binary Search Tree

The optimal Binary Search Tree problem manages to build a binary search tree with the least cost of searching a key given each key's frequency. For example, a search tree with the frequency of each key as [0.1,0.7,0.2] would have an optimal solution of 1.3 corresponding to the left one. (2)

We attack this problem by dividing BST into subtrees and adding up the cost of each subtree and the cost of the root to get the solution.

Predicate: $G[i,j] \geq \min_{i \leq k \leq j} (G[i,k-1] + s(i,j) + G[k+1,j])$ where G[i,j] refers to the cost of a subtree from index i to j, G[i,k-1] refers to the cost of left subtree given k as the root, G[k+1,j] refers to the cost of right subtree, and s[i,j] refers to the total cost added to the subtree given k as the root. This predicate ensures that the cost of each subtree equals the minimum cost of its subtrees and thus G[0, n-1] gives the cost of the optimal binary search tree.

Java Pseudocode: (3)

Table: (4)

Runtime Analysis: The code ensures that threads with different colors run sequentially, and threads with the same color run parallelly by adding the priority (j-i). Therefore, the total runtime for parallel implementation is O(n^2). In the sequential implementation, each block finds a k in O(n), and thus the runtime is O(n^3).

Matrix Multiplication:

The Matrix Multiplication problem targets finding the least computation effort of multiplying a chain of Matrices. For example, given a list of matrices M1(30*10), M2(10*30), M3(30*2), We will get 1,200 as the optimal solution (M1*(M2*M3)).

We solve this problem by dividing the chain of matrices into multiple parts. The total number of multiplications equals the sum of multiplications within each part and the computation effort of multiplying all divided parts.

Predicate: $G[i,j] \geq \min_{i \leq k < j} (G[i,k] + m_{i-1} m_k m_j + G[k+1,j]$ where G[i,j] refers to the computation effort of multiplying matrix Mi to Mj, G[i,k] refers to the computation effort of multiplying matrix Mi to Mk, G[k+1,j] refers to the computation effort of multiplying matrix Mk+1 to Mj if we split the current chain of matrices into two parts with k as the splitting point. The middle expression refers to the computation effort of multiplying the two parts. This predicate gradually increments G[i,j] to be the minimum of the computation effort summed up by its subparts. As a result, G[0,n-1] gives the optimal computation effort of this chain of matrix multiplication.

Java Pseudocode: (5)

Table:(6)

Runtime Analysis: Parallel runtime is O(n^2) and sequential runtime is O(n^3) similar to those of Optimal Binary Search Tree.

Knapsack:

The Knapsack problem, a well-known combinatorial optimization problem, challenges us to select a subset of items with given weights and values to maximize the total value of the knapsack without exceeding a given weight limit. Typically, this problem is solved using a sequential dynamic programming solution with O(nW) time complexity. In this project, we attempt to implement a parallel LLP-based solution for the Knapsack problem.

Predicate: We can model the Knapsack problem using LLP as follows. We model the feasibility as G[i, w] ≥ max(G[i - 1, w - w_i] + v_i , G[i - 1, w]) for all i, w > 0 and $w_i \leq$ w. Our goal is to find the minimum vector G that satisfies feasibility. In other words, each element G[i,w] of this vector represents the maximum value of the knapsack considering the first i items with a weight limit w. When calculating the optimality of a state, we check if the current value at G[i][w] is less than the maximum value obtainable by either including or not including the current item, based on its weight and value. If the value is not optimal, it is forbidden, thus we advance that state. Also, G[i, w] = 0 if i = 0 or w = 0 are checking if no items are considered or the knapsack has zero capacity. Each thread is responsible for calculating the optimal solution for a state (i, w) in G in parallel.

Pseudocode: (7)

Table: (8)

In the table above, the value/weight pair of an item G[i,w] represents each row and each column represents the allowed capacity of the Knapsack. The values of each row/column pair represents the optimal value of the Knapsack at that point. In this example, we illustrate a Knapsack solution given a capacity of 7 and 4 unique items to consider.

Runtime Analysis: The level of parallelism possible depends on the number of processors. Ideally, this LLP approach can significantly reduce runtime if enough processors are available to handle separate parts of the lattice concurrently. In optimal conditions, the parallel time complexity for this Knapsack solution is O(n) time and does O(nW) work, as the sequential solution to the Knapsack problem would take O(nW) time.

Conclusion:

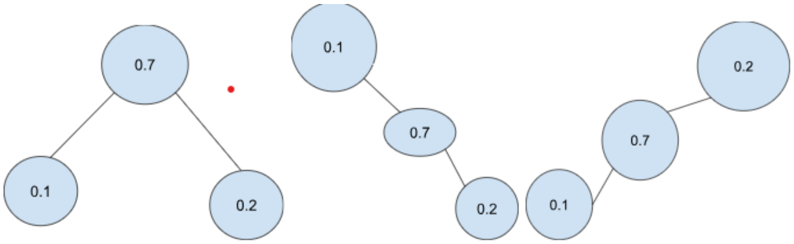
In conclusion, we did 4 parallel and 4 sequential solutions. The runtime complexity for the parallel solution is smaller, while the actual runtime for the sequential solution is smaller. (9) It's mostly because of the amount of time invested in preprocessing the parallel approach including assigning threads, calling multiple classes' methods, and initializing classes' private fields. Although the runtime for parallel approaches is longer, they might have less runtime compared to their sequential counterparts dealing with much larger datasets. Therefore, they are still useful to implement.

Reference: “*Parallel Programming” BK-Chapter 14*

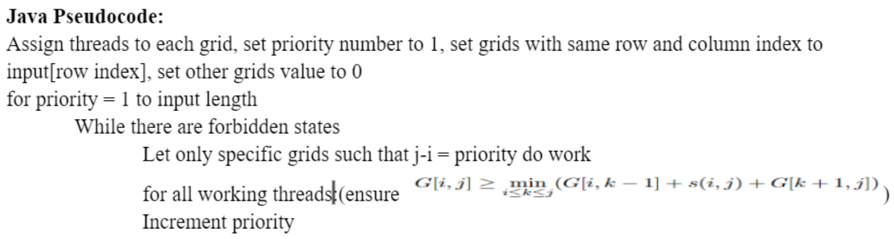
(1)

G[j]	0	1	2	3	4
	1	2	1	3	2

(2)



(3)



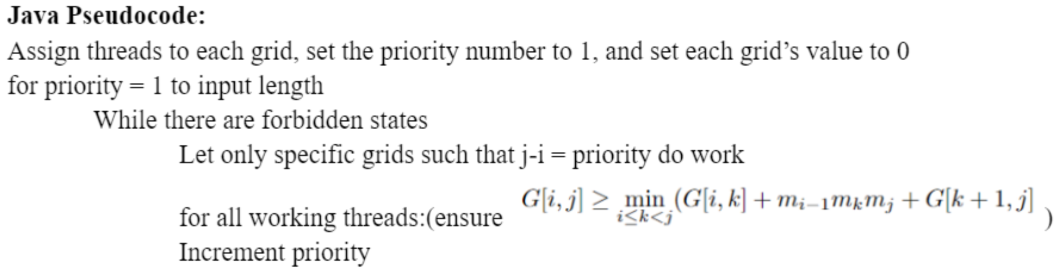
Java Pseudocode:

```
Assign threads to each grid, set priority number to 1, set grids with same row and column index to
input[row index], set other grids value to 0
for priority = 1 to input length
        While there are forbidden states
                Let only specific grids such that j-i = priority do work
                for all working threads(ensure $G[i,j] \geq \min_{i \leq k \leq j}(G[i,k-1] + s(i,j) + G[k+1,j])$)
                Increment priority
```

(4)

G[i,j] i\j	0	1	2
0	0.1	0.9	1.3
1	0	0.7	1.1
2	0	0	0.2

(5)



Java Pseudocode:

```
Assign threads to each grid, set the priority number to 1, and set each grid’s value to 0
for priority = 1 to input length
        While there are forbidden states
                Let only specific grids such that j-i = priority do work
                for all working threads:(ensure $G[i,j] \geq \min_{i \leq k < j}(G[i,k] + m_{i-1}m_k m_j + G[k+1,j])$ )
                Increment priority
```

(6)

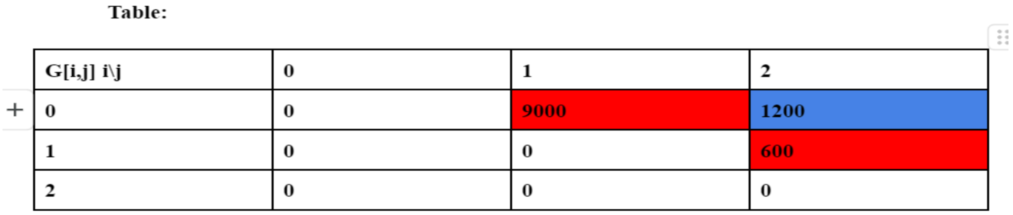
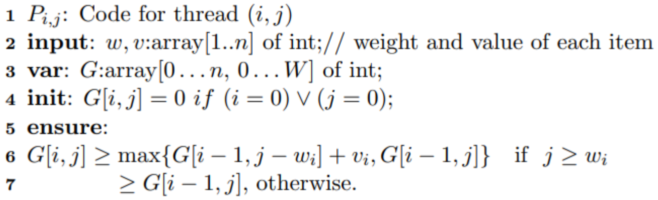


Table:

G[i,j] i\j	0	1	2
0	0	9000	1200
1	0	0	600
2	0	0	0

(7)



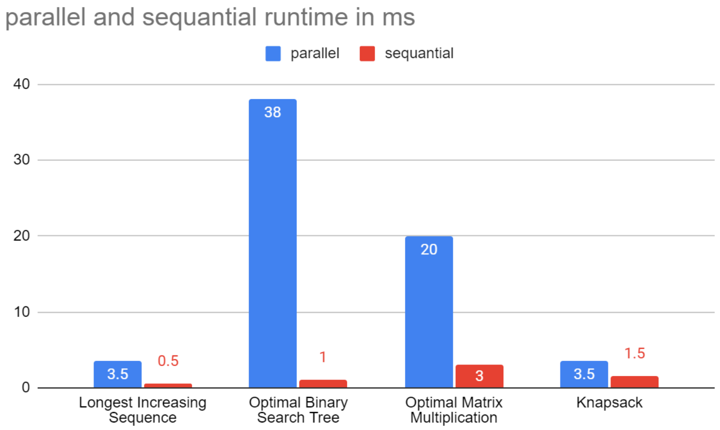
```
$P_{i,j}$: Code for thread $(i, j)$
**input**: $w, v$:array$[1..n]$ of int;// weight and value of each item
**var**: $G$:array$[0 \ldots n,\ 0 \ldots W]$ of int;
**init**: $G[i,j] = 0\ if\ \ (i = 0) \vee (j = 0)$;
**ensure**:
$G[i,j] \geq \max\{G[i-1, j-w_i] + v_i, G[i-1, j]\}$   if  $j \geq w_i$
            $\geq G[i-1, j]$, otherwise.
```

(8)

G[i, w]	0	1	2	3	4	5	6	7
1, 1	0	1	1	1	1	1	1	1
4, 3	0	1	1	4	5	5	5	5
5, 4	0	1	1	4	5	6	6	9
7, 5	0	1	1	4	5	7	8	9

(9)

Runtime Chart


parallel and sequantial runtime in ms
parallel
sequantial
Longest Increasing Sequence
Optimal Binary Search Tree
Optimal Matrix Multiplication
Knapsack