Parallel and Distributed Processing

The Knapsack Problem

Student: Boșnegeanu Adrian-Gabriel

Teacher: Lect. dr. mat. Daniel CIUIU

1. A drawing of a bag and boxes

   Description automatically generatedIntroduction

The Knapsack Problem is a classical challenge in combinatorial optimization. Given a collection of items, each characterized by a weight and a value, the goal is to determine the optimal selection of these items such that the total weight does not exceed a given limit, while the total value is maximized.

This problem dates back to the late 19th century, with early studies appearing around 1897. The name “Knapsack Problem” originates from the analogy of a traveler choosing which items to pack into a backpack without exceeding its capacity. Mathematician Tobias Dantzig is credited with popularizing the term.

The most common variant is the 0-1 Knapsack Problem, which imposes the restriction that each item can be included at most once. Given a set of n items numbered from 1 to n, each defined by a weight wi and a value vi, together with a maximum weight capacity W, the objective is to maximize the total value of selected items without exceeding the weight limit.

This problem is fundamental in optimization theory and has practical applications in various domains, such as logistics, finance, and computer science.

1. SEQUENTIAL ALGORITHM

A screenshot of a computer code

Description automatically generated

This code implements a solution for the Knapsack Problem using dynamic programming. The goal is to determine the maximum value that can be obtained from a set of items given a limited knapsack capacity.

Step-by-step explanation:

1. **Function rucsac()**:
   * Parameters:
     + cost[]: un tablou care reprezintă greutățile obiectelor.
     + valoare[]: un tablou care reprezintă valorile obiectelor.
     + n: numărul de obiecte.
     + capacitate: capacitatea maximă a rucsacului.
   * The function uses a **dynamic programming matrix (dp)** to store intermediate results
   * dp[i][w] represents the maximum value achievable using the first *i* items with a knapsack capacity of *w*.
2. **Main algorithm**:
   * **Double for-loop**
     + The first loop iterates over each item (*i* from 1 to *n*).
     + The second loop iterates over possible capacities (*w* from 1 to *capacity*).
   * If the current item’s weight cost[i-1] is less than or equal to the current capacity *w*, we can decide whether to include the item or not:
     + If included, the maximum value becomes dp[i-1][w - cost[i-1]] + value[i-1].
     + If excluded, the value remains dp[i-1][w].
   * If the item’s weight exceeds the current capacity *w*, it cannot be added to the knapsack, and the value remains the same as before.
3. **Result:**
   * At the end, the maximum value achievable with all items and the full knapsack capacity is stored in dp[n][capacitate].
4. **Function main()**:

The main() function defines item weights and values, as well as the knapsack capacity. It computes and displays the maximum possible value using the rucsac() function.

A screenshot of a computer program

Description automatically generatedIII. PARALLEL ALGORITHM

This code implements a parallel variant of the Knapsack Problem using multiple threads (multithreading). The algorithm divides objects into ranges and processes each range into a separate thread to optimize performance. Here is the detailed explanation:

**1. Function procesare\_interval():**

This is the function that processes a range of objects to find the maximum possible value in a subset of objects in that range.

* **Parameters**:
  + cost: A vector that contains the weights of objects.
  + valoare: A vector that contains the values of objects.
  + start și end: Indicates the range of objects that will be processed in this thread.
  + capacitate: Maximum backpack capacity.
  + valoare\_max\_local: will contain the maximum value found for that range of objects.
* **How it works**:
  + Generate all possible subsets of objects for the specified range.
  + Each subset is represented by a number in binary (the subset is determined by bits that are set to 1).
  + For each subset, the sum of the weights and values of the included objects is calculated. If the total weight is less than or equal to the capacity of the backpack, the total value is compared to the maximum local value found so far.
  + valoare\_max\_local is updated with the maximum value of the valid subsets (which do not exceed the capacity of the).

**2. Function main():**

This function manages the main logic of the program, including creating threads and coordinating them.

* **Initialization**:
  + Define the weights of the objects (cost) and their values (value).
  + The number of objects (n).
  + Find out the number of threads available on the current machine, using thread::hardware\_concurrency(). This will determine how many threads will be created.
* **Dividing objects between threads**:
  + The size of each range of objects that will be processed by each thread is calculated. The size is determined so that all objects are evenly distributed among the threads.
  + Each thread will process a range of objects (from start to end).
* **Creating and launching threads**:
  + For each execution thread, a thread is launched that will execute the function procesare\_interval() for the corresponding object range.
  + We use cref to pass constant references for vectors cost and valoare, again ref to be able to modify the variable valoare\_max\_local in each thread.
* **Waiting for threads to complete**:
  + After launching the threads, the program waits for each thread to finish its execution using join(). This is done for each active thread.
* **Calculating the global maximum value**:
  + After all threads have finished, the global maximum value is calculated by iterating through the local maximum values returned by each thread. Select the maximum value from them.
* **Display the result**:
  + At the end, the maximum value that can be obtained is displayed.

**3. What this algorithm does?**

* **Parallelization**: Code divides objects into multiple ranges and processes them in parallel using threads. This is a method to speed up the process, as each thread can calculate the maximum value for a subset of objects in a range.
* **Subsets of objects**: The algorithm explores all possible subsets of objects in each range, and for each subset it calculates the total value and weight. If the weight does not exceed the capacity of the backpack, the total value is compared and updated.
  1. **Results**

Sequential algorithm

* Complexity:
* Running time(10 objects, 30 capacity): **520 microseconds**

Parallel algorithm

* Complexity: **,** where p is the number of threads
* Runtime (10 objects, 30 capacity, p=5): **220 microseconds**