**SECTION 1 PROBLEM STATEMENT**

The problem entails developing a parallel program capable of automatically identifying the most valuable combination of items for transportation from a dynamic warehouse. The program must ensure that the total weight of the selected items remains within the container's capacity. The priority is to guarantee accurate output within the time limit, using parallel computing such as multithreading or multiprocessing, for improved performance, with correctness as the focus over speed. The warehouse inventory comprises a variable number of items, each with its weight and value, and the program needs to compute the optimal selection within a prescribed time limit of 300 seconds. However, KNSK.exe will terminate the program if it exceeds 180 seconds.

**Bonus**: The problem may include rules specifying that certain items can be combined to enhance their value. For example, if items A and B can be combined, the rule "AB 1 34" indicates that combining A and B increases the weight by one and the value by 34. However, it's up to the program to decide whether to utilize these rules. In the output file, items can be kept separate by listing them individually, or combined items can be represented by their combined name, following the format specified in the problem file. For instance, "AB" represents the combination of items A and B, while "BA" would result in errors. The output format must precisely match the specified naming convention.

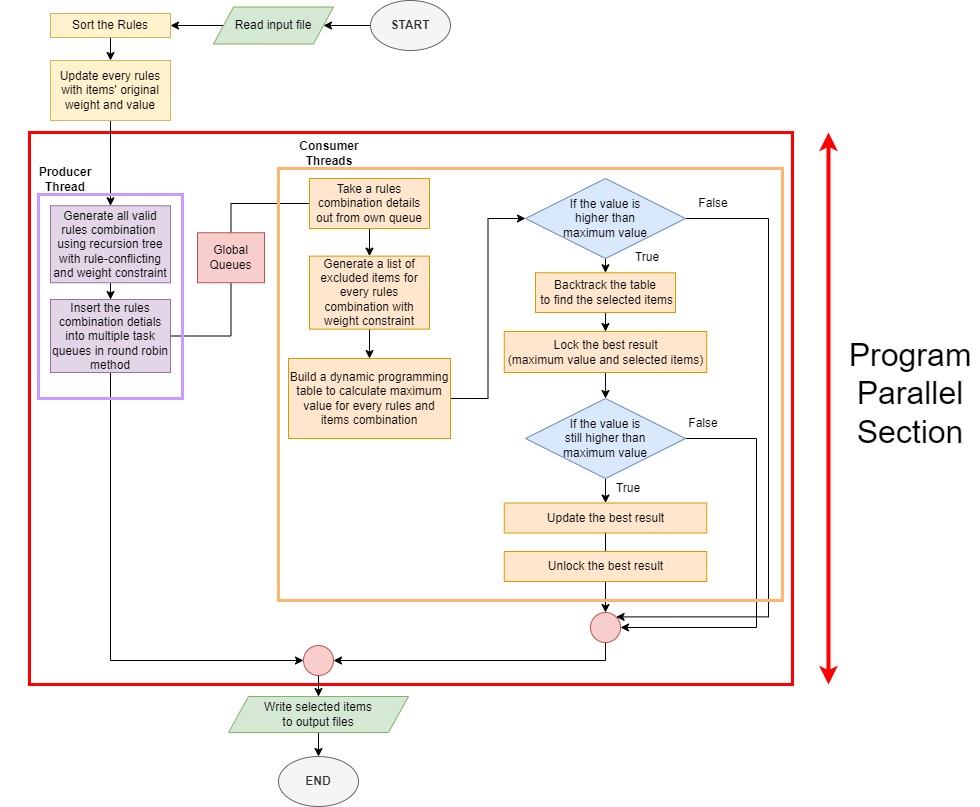
**Note**:

1) The number of CPU cores and the amount of available memory will not be disclosed in advance.

2) External libraries that require installation are not permitted.

**SECTION 2 PROGRAM FLOW**

Below is the diagram of the program flow with high level of abstraction for most of the cases.



\*The parallel section of the program will be discussed further in detail in section later.

**SECTION 3 PROGRAM PARALLEL SECTION**

The program employs the producer-consumer pattern to parallelize its execution. This section elucidates typical scenarios encountered during parallel processing. Furthermore, optimizations tailored for specific cases have been integrated and will be elaborated upon in the following section.

**Producer Thread**

**Purpose**: The producer thread is responsible for exploring all possible combinations of rules. It generates combinations for the consumer threads to calculate the local maximum value by building dynamic programming tables.

The only one producer thread relies on a **Recursive Tree with Pruning Technique**. In a naive scenario (brute-force method), where the recursion tree generates all possible combinations without constraints, the time complexity would be , where r is the number of rules, which is inefficient. To address this, two constraints are introduced during traversal of the recursion tree: the **rule-conflicting constraint** and the **weight constraint**.

The rule-conflicting constraint ensures that if the next rule conflicts with any previously selected rules, the branches (subtree) are pruned. Similarly, the weight constraint ensures that if adding the next rule would exceed the container's weight capacity, the branches are pruned as well. This pruning technique guarantees that only valid rule combinations are formed, significantly reducing the computation time and time complexity compared to .

For example, we have problem below:

Container Size: 150

Rule 1: AB with weight 70, Rule 2: BC with weight 80, Rule 3: EF with weight 90

1. Naïve recursion tree without using pruning technique:

A diagram of a diagram of a point

Description automatically generated with medium confidence

1. Recursion tree using pruning technique:

A diagram of a diagram of a diagram

Description automatically generated with medium confidence

When traversing to Point X, the program encounters a conflict when considering the inclusion of Rule 2, as both Rule 2 and Rule 1 contain character A. Consequently, the program prunes this branch to maintain consistency and avoid conflicts.

When traversing to Point Y and Point Z, the program identifies that including Rule 3 would surpass the container size limit of 150, based on the remaining weight of the branch (current rules combination). Consequently, the program prunes this branch to ensure adherence to the weight constraint.

As a result of these pruning actions, the number of valid rules combinations, considering both the rule-conflicting constraint and the weight constraint, decreases from the original 2r=23=8 to only 4 branches. This example, while illustrating the concept with only three rules, shows that as the number of rules increases, the impact of early pruning becomes more significant.

**Consumer Threads**

**Purpose**: The consumers threads is responsible for computing the local maximum value for each subset of rule combinations provided by the producer thread. These threads utilize **2-Dimensional Dynamic Programming Table** techniques to efficiently evaluate the combinations and determine the optimal selection of items.

Upon the producer thread reaching the base case, where all rules have been considered (either included or excluded), a valid rules combination with rule-conflicting and weight constraints has been established. At this juncture, the consumer thread constructs a 2D dynamic programming table using items that are excluded from the current rules combination. These items are selected based on criteria ensuring that each item's weight does not exceed the remaining capacity of the container after subtracting the total weight of the rules combination.

Subsequently, the consumer thread calculates the maximum value achievable by the current rules combination with the excluded items. If this calculated value surpasses the previous maximum value, it backtracks the 2D table to identify the items contributing to the maximum value. Finally, it updates the maximum value and the selected items in the global context.

For the knapsack problem solving idea, the following resources are referred:

1. "0/1 Knapsack Problem (Dynamic Programming) | DP-10" on ‘GeeksforGeeks’, which presents the bottom-up approach for building the dynamic programming table to maximize the knapsack's value.

*https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/?ref=lbp*

Let

1st item with weight: 1, profit: 10,

2nd item with weight:2, profit:15,

3rd item with weight: 3, profit: 40.

A screenshot of a number grid

Description automatically generated

A. Initialization: A table is set up to keep track of the maximum value achievable for different combinations of items and knapsack weights.

B. Table Construction: The table is filled up one cell at a time, starting from the top-left corner. For each cell in the table, the program considers whether it's beneficial to include the current item or not, based on its weight and value. For each item and each possible knapsack weight:

1. If it's the null item or if there's no space left in the knapsack (weight is 0), the maximum value achievable is 0 because no items can be included.
2. If the current item's weight is less than or equal to the current knapsack weight, the program has a choice (picks the option that gives the higher value):
3. It can include the current item in the knapsack. In this case, it adds its value to the maximum value achievable with the remaining space in the knapsack after adding this item.
4. It can choose not to include the current item. In this case, the maximum value remains the same as what was calculated without considering this item.
5. If the current item's weight is more than the current knapsack weight, it cannot be included, so the maximum value remains the same as what was calculated without considering this item.

C. Result: After filling up the entire table, the bottom-right cell of the table holds the maximum value achievable with the given knapsack capacity and items.

1. "Printing Items in 0/1 Knapsack" also on ‘GeeksforGeeks’, which outlines the technique for backtracking within the knapsack table to determine the items included in the optimal solution.

*https://www.geeksforgeeks.org/printing-items-01-knapsack/?ref=lbp*

A diagram of a number and a line

Description automatically generated with medium confidence

1. The backtracking process starts from the bottom-right corner of the dynamic programming table.
2. If the value in the current cell is the same as the value in the cell above it, it means that selecting the item corresponding to this row will not increase the total profit. Therefore, this item is skipped and move to the cell above it.
3. If the value in the current cell is different from the value in the cell above it, it means that selecting the item in this row will increase the total profit. Therefore, this item is selected, deduct its weight from the available container size, and move to the cell above the row with the remaining weight.
4. The process continues repeatedly until the null row is reached.
5. Once the null row is reached, the backtracking process stops, and the items that maximize the total profit is determined.

Certain modifications have been made to tailor the implementations to fit the specific needs and optimizations of the program.

1. The referred approach explicitly assigning initial values of 0 to the dynamic programming table when the row index or weight is zero. However, in the actual program, this step is skipped, relying instead on the default initialization of dynamic vectors, which sets all elements to zero.

*\* Referred code:*

**for** (i = 0; i <= n; i++) {

**for** (w = 0; w <= W; w++) {

**if** (i == 0 || w == 0)

                 K[i][w] = 0;

**else** **if** (wt[i - 1] <= w)

               K[i][w] = max(val[i - + K[i - 1][w - wt[i - 1]],

                              K[i - 1][w]);

**else**

                 K[i][w] = K[i - 1][w];

         }

     }

1. In backtracking, the check for res > 0 was skipped. The program will always proceed to the first item row of the dynamic programming table, as it doesn't include items with weights exceeding the remaining weight. This trims computational time slightly.

*\* Referred code:*

**for** (i = n; i > 0 && res > 0; i--) {

**if** (res == K[i - 1][w])

**continue**;

**else** {

              cout<<" "<<wt[i - 1] ;

             res = res - val[i - 1];

             w = w - wt[i - 1];

         }

}

**SECTION 4 PROGRAM OPTIMIZATION**

1. **Leveraging indexed item storage for faster rule updates**

The introduction of the 'items26' vector enhances rule update efficiency by sacrificing storage space for quicker item access. Organizing items based on their character index eliminates the need for nested loops during updates enabling direct mapping of each item to its corresponding index in 'items26'. It reduces complexity from two loops—one for rules and another for item searching—to a single loop solely for rules, this improves efficiency.

1. **Updating rules with items’ weights and value before recursion happens**

The process of updating rules with items' weights and values can be consolidated to avoid redundant calculations. Rather than recalculating this information repeatedly within recursive functions, it can be performed once before any recursion occurs. In this update, individual items' weights and values are updated into the rules combination before selecting items based on the rules. By consolidating these tasks into a single operation, the code significantly enhances overall efficiency.

1. **Sorting the rules before the formation of recursion tree**

A sorting step was introduced for the rules before constructing the recursion tree, employing C++'s default sort function. This sorting operation, executed in O(n log n) time where n represents the number of rules, expedites the pruning process, especially in handling conflicting rules. Given the maximum number of rules capped at 324, this approach significantly accelerates the algorithm's performance during conflict resolution.

1. **Pruning subtrees when conflict meets**

The implemented pruning technique significantly enhances the efficiency of the rule selection process by minimizing unnecessary computation. The pruning optimization is executed based on two key conditions:

1. Conflicting Rules: If selecting a rule conflict with a previously chosen one.
2. Weight Constraint: If the remaining weight is insufficient for the next rule.

By identifying and disregarding these conflicting branches, the algorithm saves both memory and computation time, as it avoids exploring paths that would lead to invalid solutions.

1. **Efficient Conflict Checking Utilizing Bit masking**

To enhance conflict checking, the program employs bit masking to represent characters in the selected rules efficiently. By utilizing a Bitmask type with 26 bits, each representing whether a character has been selected or not, the code achieves efficient storage and processing of selection information. Bit masking enables faster comparison and updates of the chosen characters, enhancing the overall efficiency of the algorithm. Instead of iterating over the entire rule combination, it swiftly verifies if the characters of the current rule intersect with the chosen characters set. This minimizes computational overhead, improving efficiency considerably.

1. **Recursion base case optimization**

In optimizing the recursive function for generating valid combinations of rules and calculating maximum value, crucial parameters such as chosen characters and remaining weight are passed down the recursion tree. This efficient approach enables seamless utilization of these values directly at the base case, eliminating redundant computations upon return to the main method. The returned result from the recursion can seamlessly pass into the knapsack function for further processing.

1. **Consider all possible case to avoid unnecessary computation**

During the calculation of the maximum value, the system considers all possible cases to avoid unnecessary computations. By exploring all possibilities, the system can do upfront calculations for these possible case and improve efficiency.

If (no rule is chosen):

Else: (at least one rule are chosen)

If (no more remaining weight for individual items):

If (there are still remaining weight for individual items):

If (no items left can be fit in the remaining container space):

Else if (only one item can be fit in the remaining container space):

Else: (more than one item can be fit in the remaining container space)

1. **Early elimination of incompatible items.**

In generating the ‘itemsLeft’ vector, the program not only verifies whether the items are already in the rules combination but also checks if their weights are less than or equal to the remaining weight of the rules combination before including the items in the dynamic programming table. The optimization using this weight constraint significantly reduces the time spent on generating the dynamic programming table, as fewer items are considered for each combination. The overall combination time is minimized, leading to improved efficiency in solving the knapsack problem.

1. **Backtrack only if necessary**

After constructing the knapsack dynamic table, a backtrack operation is initiated only if the current combination's value exceeds the current maximum value. If the achieved value is lesser, the backtrack operation is skipped to avoid unnecessary computation. This strategy optimizes the algorithm's efficiency by selectively invoking the backtrack process based on the comparison of values, ensuring optimal results while minimizing computational overhead.

1. **System-level optimization**

Employing system-level optimization techniques involves elevating the priority class of the program to a real-time level. By doing so, the program receives immediate attention from the system scheduler, minimizing context switching and reducing interruptions from other system tasks. This prioritization allows the program to execute swiftly and efficiently, enhancing its performance.

1. **Minimzed Locking for Updating Best Result**

This optimization reduces lock time on the maxValue and global selectedItems vector. Instead of directly locking the maxValue and selectedItems and pushing items and rules one by one which take longer time, this method first pushes them into a separate local vector. Locking only occurs during the updateSelectedItems function when updating the best result, where both maxValue and selectedItems are locked. Before updating, it locks both variables and checks if the local value is still greater than maxValue. If it is, only then does it update maxValue and selectedItems by changing the pointer to local vector. This update operation is swift and efficient, involving only pointer changes rather than modifying individual elements.

1. **No Locking for Backtracking**

The optimization strategy revolves around avoiding locking the global maxValue variable during the backtracking phase of the dynamic programming table. Instead of locking the maxValue during this process, the program reads it and performs backtracking if the local value exceeds the maxValue. This strategy serves to mitigate the need for long time locking during the intensive backtracking process, which can potentially lead to performance bottlenecks and thread contention. While this approach may result in some unnecessary backtracking in cases where other threads update maxValue to be greater during the backtracking process, the benefits of minimizing the time spent locking maxValue outweigh the potential inefficiency of occasional redundant computations.

1. **Individual Consumer Queues**

Multiple consumer threads concurrently process tasks generated by the producer thread. Each consumer thread possesses its own task queue, and the producer assigns tasks to these queues in a round-robin fashion. This optimization evenly distributes the workload among the consumer threads, mitigating contention of the consumer threads for accessing shared resources. By allocating tasks to distinct queues, synchronization between consumer threads is obviated when retrieving tasks from their respective queues, ensuring queue isolation. The only synchronization points arise during task enqueueing by the producer and dequeuing by the consumers, substantially reducing the overhead associated with synchronization.

1. **Eliminating Busy Waiting with Conditional Variables**

The conditional variable (cv) optimizes the producer-consumer pattern by facilitating efficient communication between producer and consumer threads. Instead of continuous polling, consumer threads enter a waiting state using cv.wait(), atomically releasing the mutex and sleeping until notified by the producer. This eliminates busy waiting, conserving CPU and reducing contention for shared resources. When the producer enqueues a task, it calls cv.notify\_one(), waking up a waiting consumer thread. This ensures active consumer threads process tasks, avoiding unnecessary resource consumption by idle threads.

**SECTION 5 OTHER CONSIDERATIONS IN PROGRAM OPTIMIZATION**

Below are some existing and non-existing **normal cases** in the problem.txt file. Existing normal case mean those that KNSK.exe can generate the case, while non-existing normal cases are those that KNSK.exe cannot generate the cases.

|  |  |
| --- | --- |
| **Existing normal case** | **Non-existing normal cases** |
| 1. No rules 2. No input file | 1. Zero container size 2. No items and rules |

The program can gracefully handle all the cases above. Despite the absence of rules, it ensures efficient execution without requiring additional optimization. For the non-existing normal cases, although checks can be implemented immediately after reading the problem.txt file and terminate the program, such checks are not implemented because KNSK.exe does not generate such scenarios. This decision helps avoid unnecessary computation time while ensuring the program's efficiency.

Below are some non-existing **extreme cases** in the problem.txt file that KNSK.exe cannot generate.

|  |
| --- |
| **Non-existing extreme cases** |
| 1. Negative container size |
| 1. Wrong input file format |
| 1. No items but have rules |

The program does not handle non-existing extreme cases above since handling these cases would require additional computation time and could sacrifice speed, which is unnecessary since they will not be generated by KNSK.exe.

The program has been tested and observed may not to select combinations that generate lower weights while achieving the same maximum value. Although a potential solution involves sorting the items, where items with the same value but lower weight are placed at the front of the vector, and then using those items as the first parameters in the ‘max’ function within the backtracking of the dynamic programming table, such an approach would consume additional computation time. Since the problem specification does not necessitate selecting the lowest weight, the program refrains from sorting the items to prioritize speed.

**SECTION 6 OTHER EXPERIMENTAL APPROACHES TESTED**

**Approach 1: Single-Level Thread Spawning**

In attempting to parallelize the computation, the initial approach was to assign different threads to handle distinct branches of the recursion tree, with the number of threads corresponding to the number of cores available on the device. This task distribution method follows a hierarchical approach aimed at effectively leveraging available computing resources. The strategy involved distributing the workload among available threads based on the level of the recursion tree to find possible rule combinations and their corresponding values. While this approach appeared theoretically sound, practical challenges hindered its effectiveness. Upon observation, it became apparent that the unknown early pruning probability posed a significant obstacle. After several experiments, it was consistently observed that only one thread remained active at the half of execution. Each thread was assigned to process a distinct branch of the recursion tree, resulting in varying workloads among threads. However, certain threads encountered branches with significantly heavier computational requirements, leading to prolonged execution times compared to others due to early pruning. Consequently, these faster threads remained idle, waiting for the slower threads to complete their tasks before proceeding, resulting in suboptimal utilization of computational resources. Despite efforts to evenly distribute work based on tree levels, the actual workload distribution proved to be imbalanced. As a result, despite attempts at parallelization, the overall speedup fell short of expectations, with the parallelized implementation taking longer to execute compared to the sequential counterpart. This failure underscores the importance of load balancing and resource management in parallel programming.

A diagram of a thread

Description automatically generated

**Approach 2: Multiple-Level Thread Spawning**

In this approach, each thread is assigned to traverse a specific branch of the recursion tree, level of the recursion tree will spawn a thread, and each thread explores a unique path starting from that level. By allocating threads to explore different branches of the tree, the workload is divided among threads, representing different combinations of rules and calculating the maxValue directly. Threads are distributed in such a way that each enabling concurrent exploration of multiple paths in the tree. In experiments, however, we observed that despite this distribution strategy, there was consistently one thread running for a prolonged period while other threads are idle. This occurrence can be attributed to the unknown probability of early pruning, where certain branches of the tree are terminated prematurely due to specific conditions not being met. Consequently, the workload imbalance persists, with certain branches requiring more computational effort than others. As a result, some threads may become idle while others continue processing, leading to suboptimal resource utilization and potentially impacting the overall performance of the parallelized implementation.

A diagram of a triangle with circles and a check mark

Description automatically generated

**Approach 3: Parallelized Dynamic Programming Table Construction In One Shot**

The program first dedicates a single thread to compute all potential combinations of rules, efficiently generating and evaluating each combination to determine its value and remaining weight. These details, encompassing the rules combination, its chosenchars, and the remaining weight, are encapsulated within a struct. Subsequently, these structs are inserted into a vector, forming a comprehensive repository of combination details. With the combination data assembled, the program proceeds to calculate a dynamic programming table using a for loop, which likely assesses the optimal value for each conceivable weight limit based on the discovered combinations. To expedite this calculation, OpenMP is employed to parallelize the for loop, distributing the workload evenly across available threads. This parallelized approach leverages the computational power of multiple cores, accelerating the computation of the dynamic programming table and enhancing overall performance.

The method occasionally demonstrates speed enhancements by leveraging all available processing cores to concurrently construct the dynamic table. However, this acceleration is most pronounced with smaller problem sizes. As the problem size increases, so does the number of combinations to consider, leading to a higher demand for memory. Consequently, each combination necessitates a deep copy operation, rapidly saturating the device's memory. When memory capacity is surpassed, data overflow occurs, prompting the need to store data in secondary storage, resulting in slower retrieval times. As a result, overall performance suffers, particularly for larger problems. While the method proves effective for smaller problem sizes, memory overload becomes apparent at a container size of 250, hindering further speed enhancements. As the sequential program does not encounter this issue, this method will not choose as the final product.

**Approach 4: Base-Case Triggered Thread Spawning**

This method employs a single thread to calculate all possible combinations of rules. When this thread encounters a base case, it spawns another thread and perform deep copy to calculate the maximum value for that specific combination. However, the performance of this method is slower compared to the sequential approach. One significant factor contributing is deep copy and thread initialisation overhead. The number of threads spawned equals the number of valid rules combinations. As this process continues iteratively until all base cases are processed, resulting in the creation of numerous threads throughout the computation. This overhead becomes particularly noticeable as the program scales up, resulting in slower overall execution times.

**Approach 5: Combination of Approach 1 with Producer Consumer Pattern**

Incorporating elements of the initial approach with a producer-consumer pattern, this method employs four threads as producers, generated at same node level to discover potential rule combinations and create corresponding tasks. Subsequently, four additional threads operate as consumers, handling these tasks to calculate the maxValue. However, similar to the first approach, the bottleneck remains consistent. During testing, execution time increased, and only one thread remained active in later stages. This imbalance stemmed from assigning threads to different branches, resulting in uneven workloads. Consequently, faster threads idled while waiting for slower ones, leading to inefficient resource utilization. Moreover, extensive mutex usage further exacerbated this issue, making this approach slower than the initial approach.

**Section Summary:**

In short, the approaches discussed in this section were not implemented in the final product. While utilizing more cores theoretically leads to faster execution times, practical testing revealed that even with all 8 cores of the testing machine operating simultaneously, the best sequential program still outperformed these approaches. Despite being scalable, the tested approaches failed to outperform the selected producer-consumer pattern parallel program, which, while scalable to some extent, demonstrated diminishing returns beyond 2 cores where scaling up to more cores resulted in worse performance. Perhaps on a supercomputer with substantial memory resources, these alternative approaches could be reconsidered. However, at present, the producer-consumer pattern remains the most effective in terms of speedup.

**SECTION 7 PERFORMANCE ANALYSIS**

In the performance analysis section, we will compare the performance of the parallel program against the best sequential program. We will assess the performance of the parallel program based on key metrics such as execution time, speedup, efficiency, and total parallel overhead.

* The sequential program has been optimized using various techniques outlined earlier in the report to achieve maximum efficiency.
* The performance analysis will be based on a problem with a container size of 300, a maximum of 26 items, and a maximum of 324 rules, using the mean execution time obtained after conducting several experiments.
* The testing machine has a maximum of 8 cores available for conducting the performance tests.

1. **Execution Time**

The execution time graph demonstrates the relationship between the number of threads and the time taken to complete the task. As the number of threads increases, the execution time initially decreases, indicating improved performance with increased parallelism. However, beyond a certain point, adding more threads leads to diminishing returns, as evidenced by the gradual increase in execution time beyond a certain thread count. From the execution time graph:

* With 1 thread, the execution time is 25.077 seconds.
* As the number of threads increases, the execution time decreases, reaching a minimum of 17.598 seconds with 2 threads. This improvement is attributed to parallelism, with one thread dedicated to producing tasks and another for consumption.
* However, beyond 4 threads, where one thread is the producer and the rest are consumer threads, adding more threads results in increased execution time compared to the sequential program. This is evident from the gradual increase in execution time as the thread count goes from 2 to 12.

1. **Speed Up**

Speed Up Formula:

The speedup graph illustrates how the parallel program's performance scales as the number of threads increases compared to the sequential version. From the speedup graph:

* The best speedup achieved is 1.425 with 2 threads, indicating that the parallel program executes 1.425 times faster than the sequential program at this point.
* As the number of threads increases, the speedup initially rises but then gradually decreases. For example, with 3 threads, the speedup decreases slightly to 1.2159, and with 4 threads, it further decreases to 1.0405.
* Beyond 4 threads, adding more threads leads to diminishing returns that resulting in the parallel program being slower than the sequential version. This is evident from the decreasing trend in speedup as the thread count increases. For instance, with 8 threads, the speedup drops to 0.4488, indicating that the parallel program with 8 threads executes only 44.88% as fast as the sequential program.
* The speedup continues to decline beyond 8 threads, suggesting that adding more threads provides diminishing returns in terms of performance improvement.

1. **Efficiency**

Speed Up Formula:

The efficiency graph illustrates the efficiency of the parallel program relative to the ideal scenario of perfect linear speedup, where doubling the number of threads should ideally halve the execution time. From the efficiency graph:

* As the number of threads increases, the efficiency gradually decreases. For example, with 2 threads, the efficiency drops to 71.25%, indicating that each thread contributes only 71.25% of the ideal speedup, where the ideal speedup is 100% efficiency.
* Initially, the efficiency experiences a significant drop. However, as the number of threads used increases, the rate of decline in efficiency becomes lower.
* The decreasing trend in efficiency highlights the increasing overhead associated with managing additional threads.

1. **Total Parallel Overhead**

Total Parallel Overhead Formula=

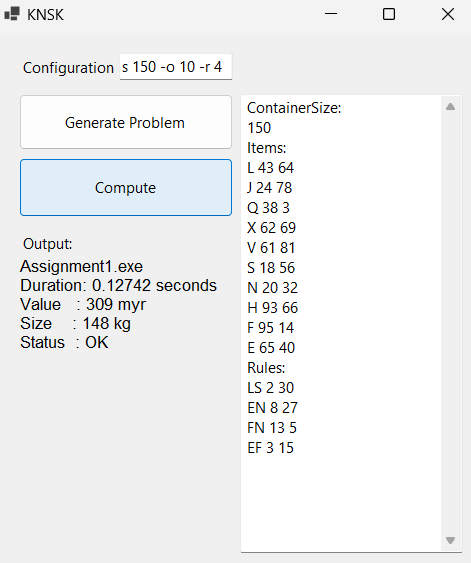
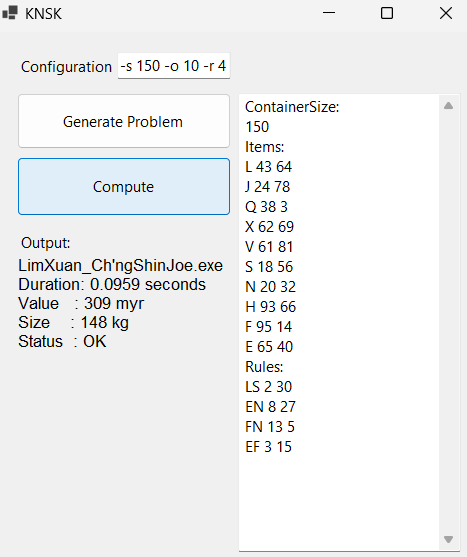
The parallel overhead graph illustrates the overhead incurred due to parallelization as the number of threads increases. From the parallel overhead graph:

* As the number of threads increases, the overhead also substantially increases. For instance, with just 2 threads, the overhead is 10.119 seconds, indicating the additional processing burden incurred by parallel execution compared to a single thread.
* The overhead rises sharply with each additional thread, reaching 421.931 seconds with 8 threads, indicating the significant amount of extra processing required to manage and coordinate multiple threads concurrently.
* Combined with the execution time graph, while adding more threads initially improves performance, the escalating overhead eventually outweighs the benefits, leading to reduced performance.

For the conclusion of the performance analysis, it is evident that the best performance, achieving a sublinear speedup of 1.45x, is obtained when using 2 threads, with one dedicated to producing and the other for consuming. However, beyond a thread count of 4, the overhead of managing additional threads surpasses the benefits of parallelism, resulting in longer execution times. Therefore, the program demonstrates improved speedup only when utilizing 2 to 4 threads, with the optimal performance observed when using 2 threads.

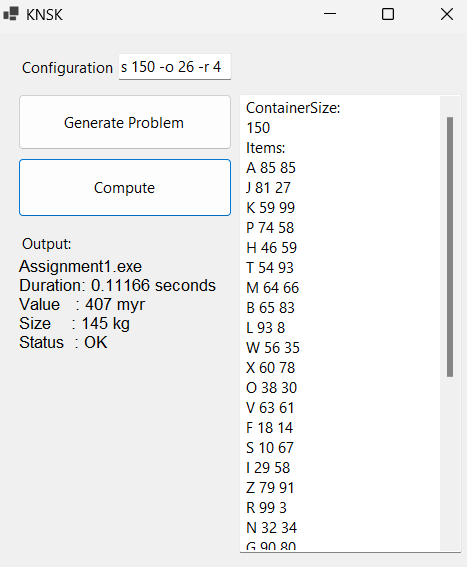
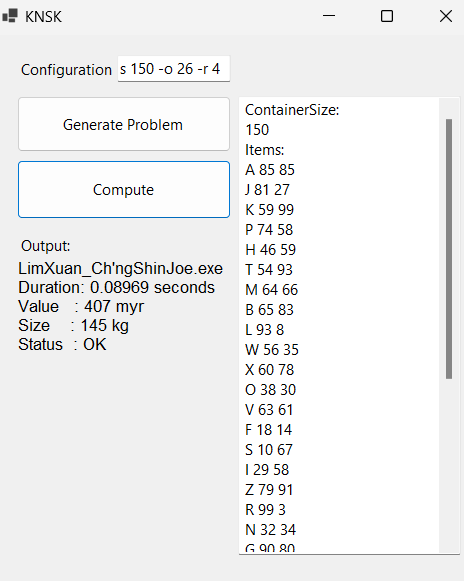
**SECTION 8 TEST CASES**

**Test Case 1: -s 150 -o 10 -r 4 (default parameters)**

** **

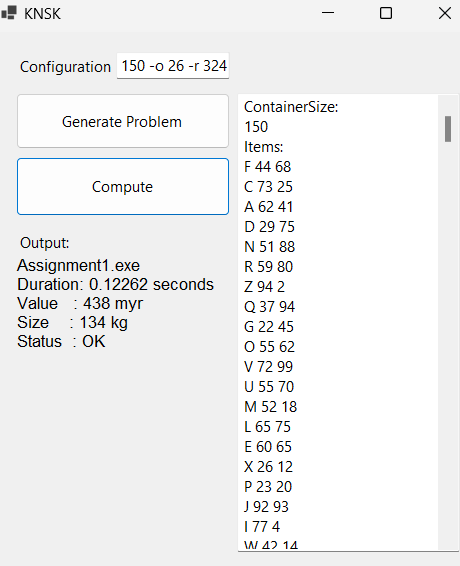
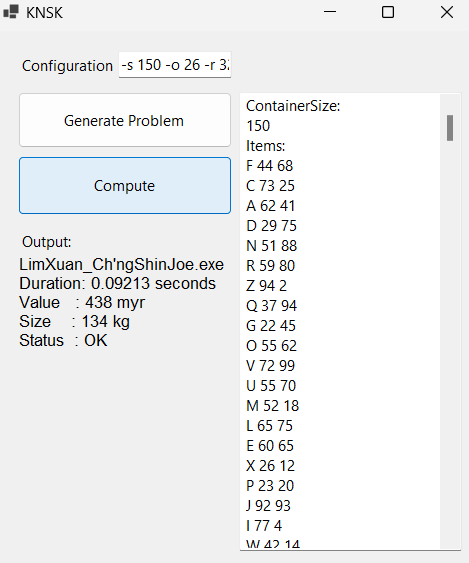
**Speedup: 1.3287x**

**Test Case 2: -s 150 -o 26 -r 0 (default container size, maximum items, no rules)**

** **

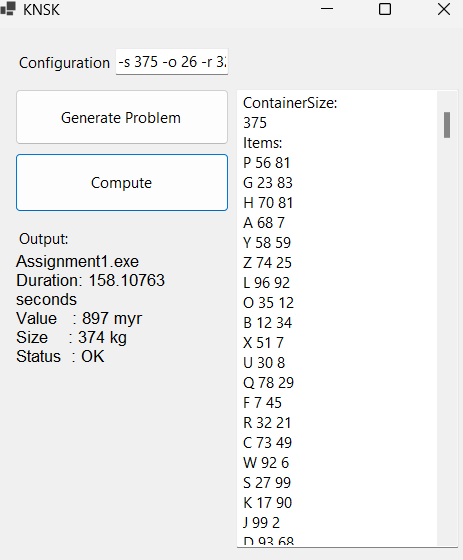
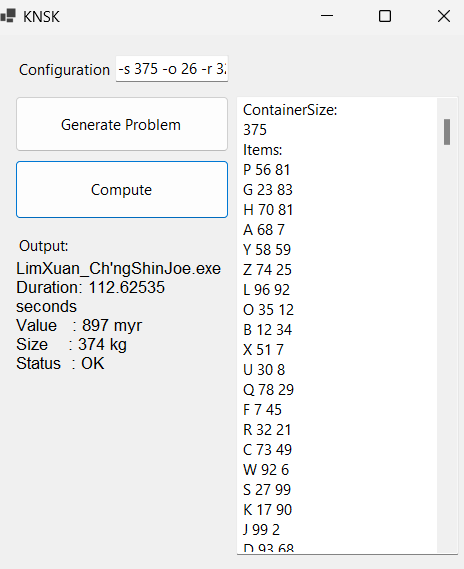
**Speedup: 1.2450x**

**Test Case 3: -s 150 -o 26 -r 324 (default container size, maximum items and rules)**

** **

**Speedup: 1.3313x**

**Test Case 4: -s 375 -o 26 -r 324 (maximum container size that can handled by this laptop, maximum items and rules)**

** **

**Speedup: 1.4107x**

**SECTION 9 CONCLUSION**

The conclusion brings to several critical aspects of the parallel program's performance and scalability. It acknowledges the achieved sublinear speedup, highlighting the 1.4x improvement, particularly notable when employing two cores, each dedicated to the producer and consumer threads, respectively. However, it also recognizes a bottleneck in the form of the producer thread's speed, hindering further scalability.

Looking forward, it outlines a pathway for future enhancements, emphasizing the need to optimize task generation for consumers. It suggests exploring more efficient load-balancing methods to distribute the task generation workload across multiple producer threads. This approach aims to mitigate bottlenecks and reduce overhead, ultimately enhancing overall program efficiency and scalability.

Furthermore, the conclusion aligns with the three fundamental criteria of good parallel programming: correctness, performance, and scalability. While correctness and performance have been largely achieved in the parallel program, scalability is acknowledged as partially accomplished. The program demonstrates scalability up to a certain point but exhibits diminishing returns beyond two threads, highlighting an area for further refinement and optimization.

**SECTION 10 ACTUAL CODE**

#include <iostream>

#include <vector>

#include <string>

#include <unordered\_set>

#include <fstream>

#include <sstream>

#include <windows.h>

#include <algorithm>

#include <queue>

#include <condition\_variable>

#include <mutex>

#include <thread>

#include <chrono>

using namespace std;

typedef uint32\_t Bitmask;

// Structure to represent an item

struct Item {

char name = ' ';

int weight = 0;

int value = 0;

};

// Structure to represent a rule

struct Rule {

string name;

int totalWeight = 0;

int totalValue = 0;

};

struct CombinationDetails {

vector<int> rulesCombination;

Bitmask chosenChars = 0;

int remainingWeight = 0;

};

// Global variables

int containerSize;

int itemsTotalWeight = 0;

int itemsTotalValue = 0;

int maxValue = 0;

vector<Item> items;

vector<Rule> rules;

vector<Item> emptyItemsCombination;

vector<int> emptyRulesCombination;

vector<string> selectedItems;

condition\_variable cv;

bool finished = false;

mutex mtxForBestResult;

vector<queue<CombinationDetails>> allCombinationDetailsQueue;

vector<mutex\*> mtxForQueues;

int queueFlag = 0;

int totalNumConcurrentThreads;

int numConsumerThreads;

//// Function prototypes

bool parseProblemFile(const string inputFile, vector<Item>& items26);

bool compareRules(const Rule& rule1, const Rule& rule2);

void updateRulesWithItemWeightsAndValues(const vector<Item>& items26);

bool newValueBiggerThanMaxValue(const int newValue);

void getNumCores();

int getBitIndex(char c);

void setBit(Bitmask& chosenChars, char c);

void unsetBit(Bitmask& chosenChars, char c);

bool isBitSet(Bitmask& chosenChars, char c);

void selectItems();

void threadsInit();

void consumer(int threadID);

void generateRulesCombinationAndCalculateMaxValue(vector<int>& rulesCombination, int index, Bitmask& chosenChars, int remainingWeight);

bool isAddingNextRuleValid(const Rule& rule, Bitmask& chosenChars);

void calculateMaxValue(const vector<int>& rulesCombination, Bitmask& chosenChars, const int remainingWeight);

vector<Item> generateItemsLeft(Bitmask& chosenChars, const int remainingWeight);

void buildDPTableAndBacktrack(const int rulesCombinationValue, const int availableWeight, const vector<Item>& availableItems, const vector<int>& rulesCombination);

void updateSelectedItems(const int newValue, vector<Item>& itemsToPush, const vector<int>& rulesToPush);

void writeSelectedItemsToFile(const string outputFile);

int main() {

// Set process priority for optimization

SetPriorityClass(GetCurrentProcess(), REALTIME\_PRIORITY\_CLASS);

// Parse the problem file

string inputFile = "problem.txt";

vector<Item> items26(26);

if (!parseProblemFile(inputFile, items26)) {

return 1;

}

// Solve single threaded if no rules

if (rules.size() == 0) {

if (itemsTotalWeight <= containerSize) {

if (!newValueBiggerThanMaxValue(itemsTotalValue)) {

updateSelectedItems(itemsTotalValue, items, emptyRulesCombination);

}

}

else {

// Solve the knapsack problem with empty rules combination

buildDPTableAndBacktrack(0, containerSize, items, emptyRulesCombination);

}

}

else {

// Sort rules based on their name

sort(rules.begin(), rules.end(), compareRules);

// Update rules with item weights and values

updateRulesWithItemWeightsAndValues(items26);

// Get hardware thread count

getNumCores();

if (totalNumConcurrentThreads == 1) { // Single threaded solution

selectItems();

}

else { // Multi-threaded solution

totalNumConcurrentThreads = 2; // After several experiment, 1+1 combination is the best

numConsumerThreads = totalNumConcurrentThreads - 1;

allCombinationDetailsQueue.resize(numConsumerThreads);

mtxForQueues.resize(numConsumerThreads);

for (int i = 0; i < numConsumerThreads; i++) {

mtxForQueues[i] = new mutex();

}

// Set up thread pool

threadsInit();

}

}

// Write selected items to an output file

string output\_file = "output.txt";

writeSelectedItemsToFile(output\_file);

return 0;

}

// Function to parse the problem file and populate data structures

bool parseProblemFile(const string inputFile, vector<Item>& items26) {

ifstream file(inputFile);

if (!file.is\_open()) {

cerr << "Error: Unable to open input file " << inputFile << endl;

return false; // Return false if file cannot be opened

}

string line;

bool parsingContainerSize = false;

bool parsingItems = false;

bool parsingRules = false;

// Read each line in the file

while (getline(file, line)) {

istringstream iss(line);

// Skip empty lines

if (line.empty()) {

continue;

}

// Identify sections in the file

if (line == "ContainerSize:") {

parsingContainerSize = true;

continue;

}

if (line == "Items:") {

parsingItems = true;

parsingContainerSize = false;

continue;

}

if (line == "Rules:") {

parsingRules = true;

parsingItems = false;

continue;

}

// Parse container size

if (parsingContainerSize) {

iss >> containerSize;

continue;

}

// Parse items

if (parsingItems) {

Item item;

iss >> item.name >> item.weight >> item.value;

items.push\_back(item);

int idx = static\_cast<int>(item.name) - 65;

items26[idx] = item; // Store items in a separate vector for faster access

itemsTotalWeight += item.weight;

itemsTotalValue += item.value;

}

// Parse rules

if (parsingRules) {

Rule rule;

iss >> rule.name >> rule.totalWeight >> rule.totalValue;

rules.push\_back(rule);

}

}

file.close();

return true;

}

// Custom comparator function to sort rules by their names

bool compareRules(const Rule& rule1, const Rule& rule2) {

return rule1.name < rule2.name;

}

// Function to update rules with item weights and values

void updateRulesWithItemWeightsAndValues(const vector<Item>& items26) {

for (Rule& rule : rules) {

// Find corresponding items in items26 vector

int idx1 = static\_cast<int>(rule.name[0]) - 65;

int idx2 = static\_cast<int>(rule.name[1]) - 65;

// Add weights of corresponding items to the rule's total weight

rule.totalWeight += items26[idx1].weight + items26[idx2].weight;

// Add values of corresponding items to the rule's total value

rule.totalValue += items26[idx1].value + items26[idx2].value;

}

}

// Function to compare value with max value

bool newValueBiggerThanMaxValue(const int newValue) {

if (newValue > maxValue) {

return true;

}

return false;

}

// Function to get total number of cores

void getNumCores() {

int num\_threads = thread::hardware\_concurrency();

if (num\_threads == 0) {

totalNumConcurrentThreads = 1; // Fallback to a single thread if unable to determine hardware concurrency

}

else {

totalNumConcurrentThreads = num\_threads;

}

}

// Function to get index for char

int getBitIndex(char c) {

return c - 'A';

}

// Function to include chosen chars

void setBit(Bitmask& chosenChars, char c) {

chosenChars |= (1 << getBitIndex(c));

}

// Function to remove chosen chars

void unsetBit(Bitmask& chosenChars, char c) {

chosenChars &= ~(1 << getBitIndex(c));

}

// Function to to check if the char is already chosen

bool isBitSet(Bitmask& chosenChars, char c) {

return chosenChars & (1 << getBitIndex(c));

}

// Function to generate valid combinations of rules using backtracking with pruning

void selectItems() {

vector<int> rulesCombination; // Current combination of rules

Bitmask chosenChars = 0;

// Start backtracking from the first rule

generateRulesCombinationAndCalculateMaxValue(rulesCombination, 0, chosenChars, containerSize);

finished = true;

cv.notify\_all();

return;

}

// Function to start all the producer and consumer threads

void threadsInit() {

vector<thread> t;

t.push\_back(thread(selectItems));

for (int j = 0; j < numConsumerThreads; j++) {

t.push\_back(thread(consumer, j));

}

for (auto& th : t) {

th.join();

}

}

// Function of consumer threads to calculate max value

void consumer(int threadID) {

while (true) {

unique\_lock<mutex> lock(\*mtxForQueues[threadID]);

cv.wait(lock, [&threadID]() {

return !allCombinationDetailsQueue[threadID].empty() || finished;

});

if (!allCombinationDetailsQueue[threadID].empty()) {

CombinationDetails combinationDetail = allCombinationDetailsQueue[threadID].front();

allCombinationDetailsQueue[threadID].pop();

lock.unlock();

calculateMaxValue(combinationDetail.rulesCombination, combinationDetail.chosenChars, combinationDetail.remainingWeight);

}

else if (finished) {

break;

}

}

}

// Function to generate valid combinations of rules with items

void generateRulesCombinationAndCalculateMaxValue(vector<int>& rulesCombination, int index, Bitmask& chosenChars, int remainingWeight) {

// Base case: All rules have been considered

if (index == rules.size()) {

if (totalNumConcurrentThreads == 1) {

calculateMaxValue(rulesCombination, chosenChars, remainingWeight);

}

else {

CombinationDetails combinationDetail;

combinationDetail.rulesCombination = rulesCombination;

combinationDetail.chosenChars = chosenChars;

combinationDetail.remainingWeight = remainingWeight;

unique\_lock<mutex> lock(\*mtxForQueues[queueFlag]);

allCombinationDetailsQueue[queueFlag].push(combinationDetail);

lock.unlock();

cv.notify\_one();

if (queueFlag == (numConsumerThreads - 1)) {

queueFlag = 0;

}

else {

queueFlag++;

}

}

return;

}

// Try not choosing the current rule

generateRulesCombinationAndCalculateMaxValue(rulesCombination, index + 1, chosenChars, remainingWeight);

// Try choosing the current rule if it doesn't conflict with chosen characters and doesn't exceed remaining weight

if (isAddingNextRuleValid(rules[index], chosenChars) && rules[index].totalWeight <= remainingWeight) {

// Add the current rule to the combination and update chosen characters

for (char c : rules[index].name) {

setBit(chosenChars, c);

}

rulesCombination.push\_back(index);

// Recur for the next rule with updated parameters

generateRulesCombinationAndCalculateMaxValue(rulesCombination, index + 1, chosenChars, remainingWeight - rules[index].totalWeight);

// Remove the current rule from the combination and restore chosen characters

rulesCombination.pop\_back();

for (char c : rules[index].name) {

unsetBit(chosenChars, c);

}

}

}

// Function to check if adding the current rule conflicts with chosen characters

bool isAddingNextRuleValid(const Rule& rule, Bitmask& chosenChars) {

for (char c : rule.name) {

if (isBitSet(chosenChars, c)) return false;

}

return true; // No conflict

}

// Function to calculate maximum value based on current rules combination and items left

void calculateMaxValue(const vector<int>& rulesCombination, Bitmask& chosenChars, const int remainingWeight) {

if (rulesCombination.size() == 0) {

// If no rule is chosen, solve the knapsack problem for items only

buildDPTableAndBacktrack(0, containerSize, items, emptyRulesCombination);

}

else {

// Calculate total value of rules combination

int rulesCombinationTotalValue = 0;

for (const int& idx : rulesCombination) {

rulesCombinationTotalValue += rules[idx].totalValue;

}

if (remainingWeight == 0) {

// If no more weight is available, add the value of rules combination to the total value

if (newValueBiggerThanMaxValue(rulesCombinationTotalValue)) {

updateSelectedItems(rulesCombinationTotalValue, emptyItemsCombination, rulesCombination);

}

}

if (remainingWeight > 0) {

// If there's still weight available, generate items left and solve knapsack problem

vector<Item> itemsLeft = generateItemsLeft(chosenChars, remainingWeight);

if (itemsLeft.size() == 0) {

// If no more items can be added, add the value of rules combination to the total value

if (newValueBiggerThanMaxValue(rulesCombinationTotalValue)) {

updateSelectedItems(rulesCombinationTotalValue, emptyItemsCombination, rulesCombination);

}

}

else if (itemsLeft.size() == 1) {

// If only one item is left, consider adding it along with rules combination

if (newValueBiggerThanMaxValue(rulesCombinationTotalValue + itemsLeft[0].value)) {

updateSelectedItems(rulesCombinationTotalValue + itemsLeft[0].value, itemsLeft, rulesCombination);

}

}

else {

// If multiple items are left, solve knapsack problem for remaining items

buildDPTableAndBacktrack(rulesCombinationTotalValue, remainingWeight, itemsLeft, rulesCombination);

}

}

}

}

// Function to generate items left based on chosen characters and remaining weight

vector<Item> generateItemsLeft(Bitmask& chosenChars, const int remainingWeight) {

vector<Item> itemsLeft;

for (const Item& item : items) {

if (isBitSet(chosenChars, item.name) == false) {

if (item.weight <= remainingWeight) {

itemsLeft.push\_back(item);

}

}

}

return itemsLeft;

}

// Function to build the DP table and perform backtracking for knapsack problem

void buildDPTableAndBacktrack(const int rulesCombinationValue, const int availableWeight, const vector<Item>& availableItems, const vector<int>& rulesCombination) {

int W = availableWeight;

size\_t I = availableItems.size();

vector<vector<int>> dp(I + 1, vector<int>(W + 1)); // Dynamic programming table

// Build dynamic programming table dp[][] in bottom-up manner

for (int i = 1; i <= I; i++) {

for (int w = 0; w <= W; w++) {

if (availableItems[i - 1].weight > w) {

dp[i][w] = dp[i - 1][w];

}

else {

dp[i][w] = max(dp[i - 1][w], availableItems[i - 1].value + dp[i - 1][w - availableItems[i - 1].weight]);

}

}

}

// Backtrack to find the selected items only if more than maximum value

if (newValueBiggerThanMaxValue(dp[I][W] + rulesCombinationValue)) {

vector<Item> itemsCombination;

int remainingWeight = availableWeight;

for (size\_t i = I; i > 0; i--) {

if (dp[i][remainingWeight] != dp[i - 1][remainingWeight]) {

itemsCombination.push\_back(availableItems[i - 1]);

remainingWeight -= availableItems[i - 1].weight;

}

}

updateSelectedItems(dp[I][W] + rulesCombinationValue, itemsCombination, rulesCombination);

}

}

// Function to push string to selectedItems vector

void updateSelectedItems(const int newValue, vector<Item>& itemsToPush, const vector<int>& rulesToPush) {

vector<string> localSelectedItems; // Local variable to hold the selected items

for (const int& idx : rulesToPush) {

localSelectedItems.push\_back(rules[idx].name);

}

for (const Item& item : itemsToPush) {

localSelectedItems.push\_back(item.name + string(""));

}

if (totalNumConcurrentThreads == 1) {

maxValue = newValue;

selectedItems = localSelectedItems;

}

else {

// Update the global maxValue and selectedItems inside a critical section

unique\_lock<mutex> lock(mtxForBestResult);

if (newValue > maxValue) {

maxValue = newValue;

selectedItems = localSelectedItems;

}

lock.unlock();

}

}

// Function to write selected items to an output file

void writeSelectedItemsToFile(const string outputFile) {

ofstream outFile(outputFile);

if (!outFile.is\_open()) {

cerr << "Error: Unable to open output file " << outputFile << endl;

return;

}

for (const string& item : selectedItems) {

outFile << item << endl;

}

outFile.close();

}