CMPT 431 – Distributed Systems
Simon Fraser University
Fall 2024

0/1 KNAPSACK – Threaded and Distributed Methods

David Krljanovic #301427415,
HoJun Lee #301359952,
Shabbir Yusufali #301414687

Introduction

The 0/1 knapsack problem is a fundamental combinatorial optimization problem where "given a set of items, each with a weight and a value, determine which items to include in the collection so that the total weight is less than or equal to a given limit and the total value is as large as possible". The "0/1" aspect implies that each item can be either excluded (0) or included (1). [1] The problem is renowned for its applicability in resource allocation, budgeting and decision-making processes. As an NP-complete problem, it presents significant computational challenges, especially as the size of the item set increases.

This report explores the performance across different computational strategies to solve the 0/1 Knapsack problem. Our team has compared three implementations: serial, parallelized using muti-threads and a distributed using Message Passing Interface (MPI). By analyzing and comparing these implementations, we were able to come up with meaningful discussion regarding the advantages and limitations of each implementation and some insights into how parallelization can enhance the performance and overall efficiency in solving real life optimization problems.

Background

The 0/1 Knapsack Problem
The Knapsack problem has been a renowned problem in algorithmic studies as it has relatively simple definition but rather complex solutions. In fact, the knapsack problem has been studied and explored for more than a century. The earliest work can be found back in 1897 from a journal article by G. B. Mathews, published by the London Mathematical Society. [2]
0/1 Knapsack problem is the most commonly solved problem among other types of Knapsack problem. 0/1 Knapsack problem restricts the number $x_i$of copies of each kind of item to zero or one. Given a set of n items numbered from 1 up to $n,$ each with a weight w_i and a value v_i, along with a maximum weight capacity $W,$ maximize $\sum_{i=1}^{n} v_i\, x_i$
subject to $\sum_{i=1}^{n} w_i\, x_i \ \leq W$ and $x_i \in \{0,\ 1\}$.

The classic solution for the 0/1 Knapsack problem uses dynamic programming, which constructs a table to store the optimal solutions for subproblems, ultimately building up to solve the main problem. [3] The time complexity of using dynamic programming is $O(nW)$.

State of Art
In serial computing, the algorithm executes sequentially on a single processor. The serial dynamic programming solution for 0/1 Knapsack problem is straightforward to implement. The serial implementation using dynamic programming is useful when the item is small in size. However, the scalability of serial approach is limited by the performance and memory of a single machine. As the problem size grows, the execution time will increase with the number of items and the capacity, ultimately causing bottleneck in performance.

To address and overcome the limitation of serial approaches, parallel approaches have been developed. The fundamental of parallel approaches in terms of 0/1 Knapsack problem is to partition the dynamic programming table and to distribute the computations across multiple threads or processes. There are two ways to parallelize the algorithm in order to solve much larger problem sizes: Muti-threading and Message Passing Interface (MPI).

Multi-threading involves the use of multiple threads within a single process, sharing the same memory space depending on the type of architecture. In the context of the 0/1 Knapsack problem, multi-threading can be applied to parallelize the populating of the dynamic programming table concurrently. As the memory space is shared, threads can directly access shared data structures reducing the need for complicated communication protocols. As it does not need to communicate between processes, rather only needs to switch between threads, it is much faster due to reduced overhead, ultimately leading to efficient resource utilization. The challenges faced using multi-threading are thread synchronization. As the consistency of dependent data is required, a thread synchronization logic needs to be made with caution. Ensuring such data consistency also may introduce overhead performance to the program, making possibility of diminishing return. To add more, as serial implementations are limited by the computation power of a single machine, multi-thread implementations are also limited by the number of cores in a single machine as the machine are distributing the workload between the threads it has.

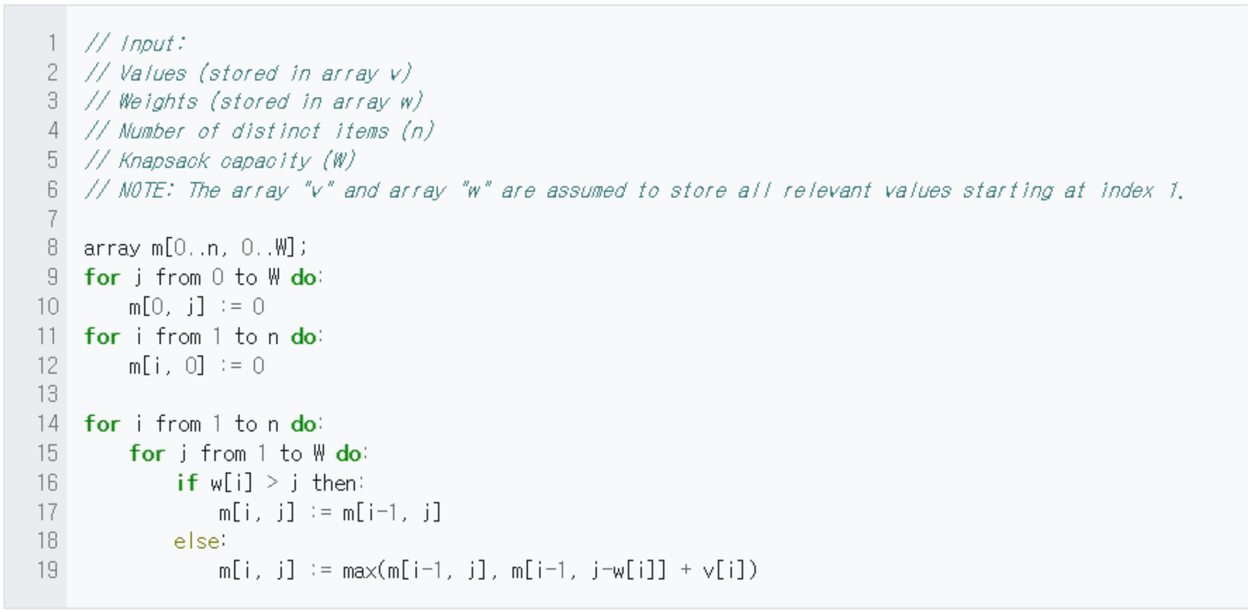
The distributed implementation using MPI is a standardized and portable message-passing system designed for parallel computing architecture. MPI allows multiple processes on different multiple machines to send and receive messages to communicate and coordinate their computations. Similar to the multi-threading implementation, in context of the 0/1 knapsack problem, the dynamic programming table is distributed amongst multiple processes to compute and populate it on their own and communicates their results to form a completed table. As MPI implementations use multiple physical machines to do computations, it can provide access to more computation resources, leading to better scalability compared to other implementations. Also, each process does not share memory with other processes, allowing each machine to handle a larger dataset that may have not fitted into memory of a single machine. However, contrary to multi-threading implementations, MPI requires a communication protocol as it has to exchange data between processes. This can create rather huge performance overhead and delay in development as efficient implementations of MPI regarding this challenge can be relatively complicated compared to other implementations.

However, solutions for the 0/1 Knapsack problem are not only limited to serial or parallel dynamic programming. As the problem existed in the industry for more than a century, other solutions have been explored as well. The two most prominent alternative approaches are "Branch and Bound Method" and the "Hybrid Method". Branch and Bound Method explores different branches of the solution space concurrently. With efficient pruning strategies, this method can be effective in reducing unnecessary computation. [4]. The Hybrid Method takes advantage of both methods, aiming to maximize resource utilization [5].

Implementation Details

Pseudocode Analysis

As discussed on the Background section of this report, serial implementation dynamic programming to solve 0/1 Knapsack problem is straightforward. For the actual implementation, I have used pseudocode from the wikipeida page of Knapsack problem [1]:



```
// Input:
// Values (stored in array v)
// Weights (stored in array w)
// Number of distinct items (n)
// Knapsack capacity (W)
// NOTE: The array "v" and array "w" are assumed to store all relevant values starting at index 1.

array m[0..n, 0..W];
for j from 0 to W do:
    m[0, j] := 0
for i from 1 to n do:
    m[i, 0] := 0

for i from 1 to n do:
    for j from 1 to W do:
        if w[i] > j then:
            m[i, j] := m[i-1, j]
        else:
            m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
```

This pseudocode serves as a foundation for all three implementations. Hence, it is important to understand what it does fully. First, a dynamic programming table m[0...n, 0...W] is created. Each cell m[i, j] will store the maximum value achieved from using the first i items with a capacity of j. The pseudocode handles two base cases at the initialization step:

- When the capacity is zero, it means the sack cannot take any items, hence the value is also zero.
- When the item is zero (i = 0), value cannot be achieved so the value is also zero.

The main loop builds the solution iteratively. For each item i and each possible capacity j, a decision is made where:

1. If the current item's weight (w[i]) is greater than our current capacity (j), the item cannot be included so the best value possible without it (m[i-1, j]) is taken instead.
2. Otherwise, the algorithm either does not take item i, which makes the value remaining m[i-1, j] or takes the item i and add its value (v[i]) to the best value possible with remaining capacity (m[i-1, j-w[i]]). These two choices are made based on the maximum value possible.

Serial Implementation

For actual implementation, some changes were made from pseudocode as design choices.
I wanted to keep Item as a struct in separate file so if there's any need for modification of Item attributes so that we could experiment with different configurations, it would be easier to. As a result, I have declared the struct in separate *Item.h* file inside core directory as follows:

```
struct Item
{
    int weight;
    int value;
    Item(int w, int v) : weight(w), value(v) {}
    Item() : weight(), value() {}
};
```

This struct was passed to the function by const reference in order to prevent copying the entire vector and to ensure the original data was not modified:

```
int knapsack_serial(const std::vector< Item > &items, int capacity)
```

A dictionary made of nested vector was used initially, but we soon found it caused a huge performance overhead. Therefore, we changed the dynamic programming table as 1D array, which we incorporated indexing rule to represent the 2D table.

```
// dynamic programing table
int *dp = new int[(n+1) * (capacity+1)]();
```

However, using such indexing was very counterintuitive. So to improve the readability of the codes, macro was defined:

```
// MACRO so that the array indexing is more readable
    #define DP(i, j) dp[(i) * (capacity+1) + (j)]
```

The rest of the implementations generally follows the pseudocode explained above. One last change made from the pseudocode was the index management. C++ uses 0-based indexing. However, to account for empty subsets, our dynamic programming dictionary uses 1-based indexing. To make sure mathematical correctness of algorithm is kept, I have adjusted the index when accessing the item as follows:

```
if (items[i-1].weight <= j)
```

Other two implementations also follow the above-mentioned design choices except the index managements for overall code coherence.

Parallel Implementation

The Parallel Implementation was a challenge given that the values in the dynamic programming table had data dependencies on other values within the table that needed to be carefully considered. More specifically, if the value we want to calculate is DP[i][j], we need to have access to all values with indices {i-1, 0 to j inclusive}. We quickly discovered that our goal was to limit the amount of idle time that threads needed to spend in order for all data to be ready for computation. By sharing the dynamic programming table among all threads, we would also eliminate the need for communication. Initial implementations featuring mutex locks were not outputting promising results, so we would have to find another way to solve data dependencies and races. Our current implementation of a parallel 0/1 knapsack algorithm features a combination of input parameter analysis, load balancing, spatial locality, and synchronization primitives in the form of barriers. There are two functions that can do work on the input data provided to the program based on the sizes capacity and item parameters passed.

The first function, row_knapsack_function, works on data that is portioned among threads by capacity; each thread will work on a separate range of weights. The logic of each thread, when compared to the serial implementation, is nearly identical. For each incrementation of the row index, the arbitrary (but equally sized) ranges of column values are calculated, and following the completion of the inner loop that performs these calculations, the threads are all kept synchronized by a CustomBarrier barrier. The barrier ensures that, on starting a new row, the values have access to the entire row of values above without having to wait for incomplete or missing values. This function will only run when the size of input capacity exceeds the number of objects. This is because the outer loop, based on the row index, controls how many barriers the threads will need to synchronize at in order to complete. If there are less item rows than there are capacity columns, then incrementing the outer loop by row will force less barriers on the overall procedure.

The second function, column_knapsack_function, is more complex and considers a scenario in which the number of items exceeds the capacity. In this case, if we iterate the outer loop by the number of rows, then we would be using more synchronization primitives than if we iterated by capacity. Our first, more naïve version of this implantation simply swapped the order of the loops — Following an even portioning of *items* (not weights) among threads, the outer loop now iterated by column index, and the inner loop iterated by row index. Just as the first function did, our threads synchronized via barrier following the completion of the inner loop. This version of the function worked, but there was no speedup in comparison to the serial implementation. In fact, it was noticeably slower! Although at first, we were puzzled, it quickly dawned on us that our order of data structure accesses were not optimized for spatial locality. In order to take advantage of caches incurred by spatial locality, we had to exchange items with weights, so that iteration by item was contiguous in memory.

Distributed Implementation

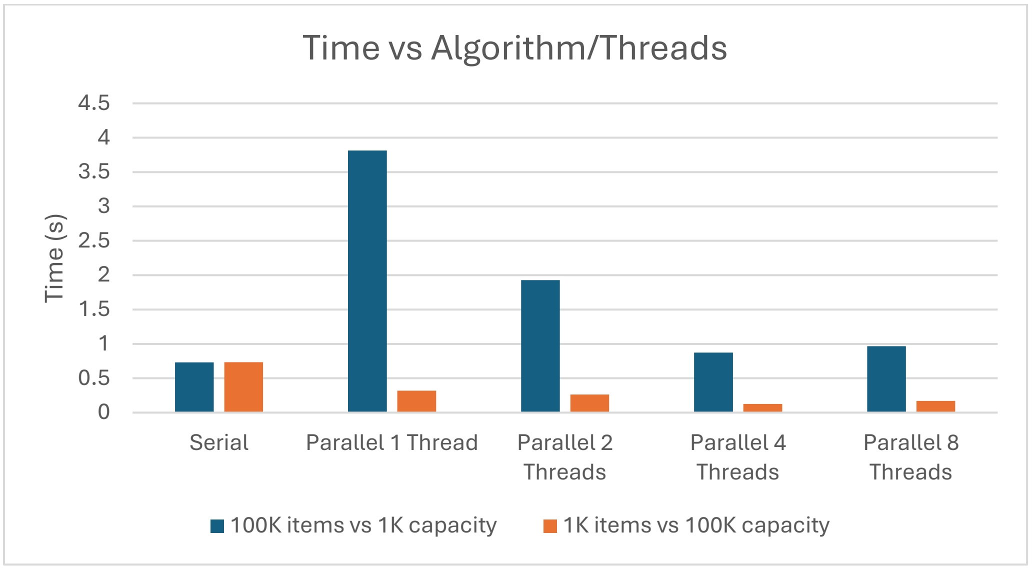
Distributed implementations of algorithms differ from typical parallel implementations – There is no notion of a shared data structure, individual processes can only access computations and calculations completed foreign processor computations by invoking communication methods. Using MPI (Message Passing Interface), we can have each process exchange values with the processes that need them. Our distributed algorithm handled load balancing by partitioning the total set of items amongst each process. Each process handled a range of items that did not overlap. The outer loop of the distributed algorithm iterated by weight, and the inner loop iterated by item. In order to save memory, each process only contained enough elements to store the values that it was responsible for calculating, as well as an extra row on the top to store the values it needed from other processes.

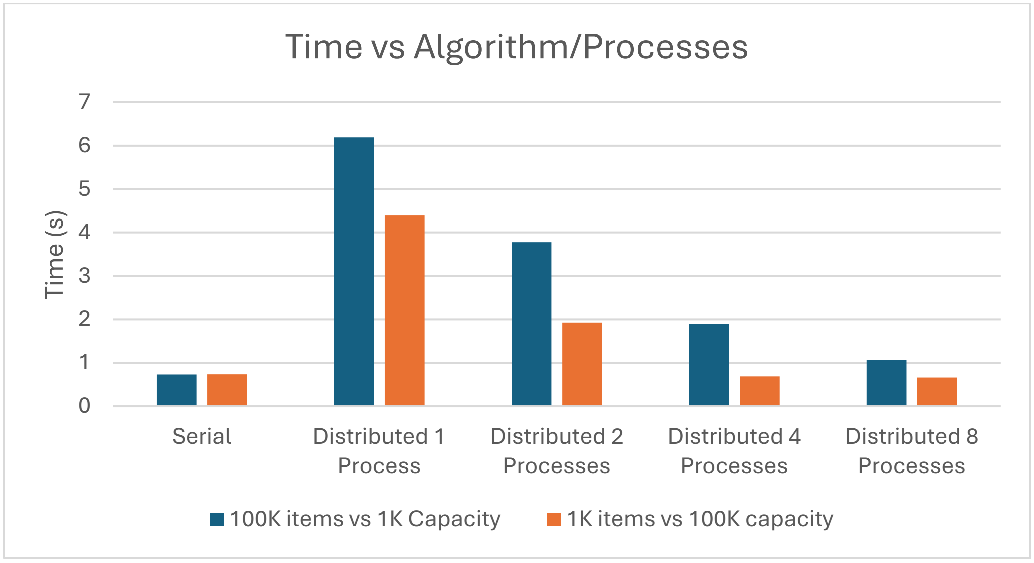
Just like in all other implementations, the calculations of each value dependent on the data in the row above. However, without a shared data structure, these values were not always conveniently available. Therefore, before the iterations of the inner loop began, processes with ranks greater than 0 waited to receive data via MPI_Recv from the process above it at the boundary of their data structures. Following the completion of the inner loop, each process with process rank less than the total amount of processes minus one would send data via MPI_Send to the process below. The data that was sent and received were sub arrays of size chunk_size. Chunk_size was defined by the total capacity divided by 10. By setting a chunk of data to be sent and received, rather than single values, we were able to cut back on communication overhead. At the end of the algorithm, values were collected via MPI_Reduce using the MPI_MAX operator.

Evaluation and Analysis

The data below was gathered by running 2 different tests on slurm 3 times for serial and parallel and distributed with 1, 2, 4 and 8 threads/processes each. The mean of the result was used

Results





All of these times were determined running on the cluster we used for assignments. Based on the runtimes (in seconds), we can see that for both the parallel and distributed implementations, the functions perform faster when capacity exceeds the number of items. However, the computations involving an excess in items were beginning to approach similar performance as the number of active threads or processes increased. For distributed processes, these results make sense: The higher the capacity, the more communication there will need to be in order for every weight column element to be filled. However, the stunted parallel performance is quite perplexing. This may be due to further locality issues that we are unable to rectify. The exact source of this slowdown among these tasks is difficult to pinpoint, considering that timings among threads and processes were very even. We were, however, very pleased to see a strong speedup among high-capacity tasks for parallel processes, roughly 6 for 8 threads! Perhaps given more time, we could find more methods to reduce idle time incurred by communication and synchronization.

Limitations

There are two limitations with our code, one is programmer defined.

- Most systems are unable to provide the memory requirements demanded by the program when in use with large inputs.
- In our distributed knapsack function, capacity is limited to multiples of ten. This is done for simplicity's sake.

Conclusions

Creating parallel and distributed solutions to the 0/1 knapsack problem was far more challenging than we anticipated. Combining almost all of our accumulated knowledge from this fall's offering of CMPT 431, we were able to achieve a speedup of approximately 6 among parallel tasks with capacities exceeding the number of items. We were very happy with this result. Although we were only *close* to achieving speedups in other categories as well, we were happy to see results that, at the least, weren't slow considering terrible performances in our initial implementations of these algorithms. Perhaps, what we have not wanted to admit thus far, is that maybe there is no greater efficient parallel implementation of this problem. Although we may have been burdened with a deadline, we will continue to research and experiment with other methods of attaining speedup out of sheer curiosity.

Reference

[1] *Knapsack problem,Wikipedia*. Available at: https://en.wikipedia.org/wiki/Knapsack_problem (Accessed: November 2024).

[2] Mathews, G.B. (1896) 'On the partition of numbers', *Proceedings of the London Mathematical Society*, s1-28(1), pp. 486–490. doi:10.1112/plms/s1-28.1.486.

[3] Andonov, R., Poirriez, V. and Rajopadhye, S. (2000) 'Unbounded knapsack problem: Dynamic Programming Revisited', *European Journal of Operational Research*, 123(2), pp. 394–407. doi:10.1016/s0377-2217(99)00265-9.

[4] Martello, S. and Toth, P. (1990) *Knapsack problems. algorithms & computer implementations*. Chichester: John Wiley & Sons Limited.

[5] Martello, S. and Toth, P. (1984) 'A mixture of dynamic programming and branch-and-bound for the subset-sum problem', *Management Science*, 30(6), pp. 765–771. doi:10.1287/mnsc.30.6.765.