Serial, Parallel, and Distributed Algorithm Performance for the 0-1 Knapsack Problem

Table of Contents

Introduction

The knapsack problem consists of a bag (or "knapsack") and some n number of items, i, with known weights, s_i. Each item also has a known value, v_i, which is a measure of some sort of profit. The bag has a maximum limit, S, for the weight it can carry. The goal is to find a set of items to fit in the bag such that the total value of those items is maximized. The most commonly addressed version of this problem is the 0-1 knapsack problem, where only zero or one copy of each item exists. The weights and values are also assumed to be positive.

Background

Since the problem entails finding the optimal combinations of items, a dynamic programming solution is best-suited here. The bottom-up approach uses a table to store previous calculations for quick recall. Specifically, for a matrix dp, $d[i][j]$ holds the maximum value generated from a subset of $i,...,n-1$ items with a maximum weight of j. The calculation of the optimal value of $d[i][j]$ via a Bellman equation comes in use for calculations in the $i+1$ row.

$$dp[i][j] = \max \begin{pmatrix} dp[i+1][j] \\ dp[i+1][j-s_i] + v_i \quad \text{if } j \geq s_i \end{pmatrix}$$ [1]

The running time of this algorithm is $O(n*S)$, corresponding to the size of the matrix dp. This is much more efficient than a brute-force approach which is $O(2^n)$, as there are 2^n possible combinations of items.

Implementation Details

The terms "*dp* matrix" and "dynamic table" are used interchangeably in the paper to refer to the table used to store calculations in the dynamic programming algorithm.

I. Serial

The iterations begin from the n^{th} row which is known to evaluate to all zeroes because no items from the subset $n,...,n-1$ can be added to the bag. Then from row $n-1$ onwards, the Bellman equation is computed for all pairs (i, j).

[1] https://courses.csail.mit.edu/6.006/fall11/rec/rec21_knapsack.pdf

```
KNAPSACK(n, S, s, v)
 1  for i in {n, n − 1 ... 0}
 2      for j in {0, 1 ... S}
 3          if i == n
 4              dp[i][j] = 0 // initial condition
 5          else
 6              choices = []
 7              APPEND(choices, dp[i + 1][j])
 8              if j ≥ s_i
 9                  APPEND(choices, dp[i + 1][j − s_i] + v_i)
10              dp[i][j] = MAX(choices)
11  return dp[0][S]
```

II. Parallel

The parallel version runs the serial knapsack function with C++ threads. Each thread operates on a subset of the columns *0,...,S*. The sub-matrix that each thread, *t*, calculates is of size $n * m_t$, where

$$m_t = \begin{cases} (S+1)/\text{number of threads} + 1 & t < rem \\ (S+1)/\text{number of threads} & t >= rem \end{cases}$$

$$rem = (S+1) \mod (\text{number of threads})$$

A barrier placed at the end of each row iteration synchronizes the threads to ensure that row *i* is complete before calculating row *i-1*.

III. Distributed

The MPI implementation of the algorithm uses the same algorithm as the parallel version but with a different method of distributing the data. Instead of partitioning the matrix into blocks, the following equation is iterated on:

$$next\ column\ number\ for\ process\ i\ =\ worldRank(i)\ +\ worldSize$$

Each process needs to send its calculation of *d*[*i*][*j*] to the process operating on column __ and needs to receive data from the process that calculated *d*[*i+1*][*j-s*[*i*]]. Additionally, each process has to send all its calculated values to the root process. The root process uses the complete *dp* matrix to produce a list of items that are included in the knapsack.

Evaluation

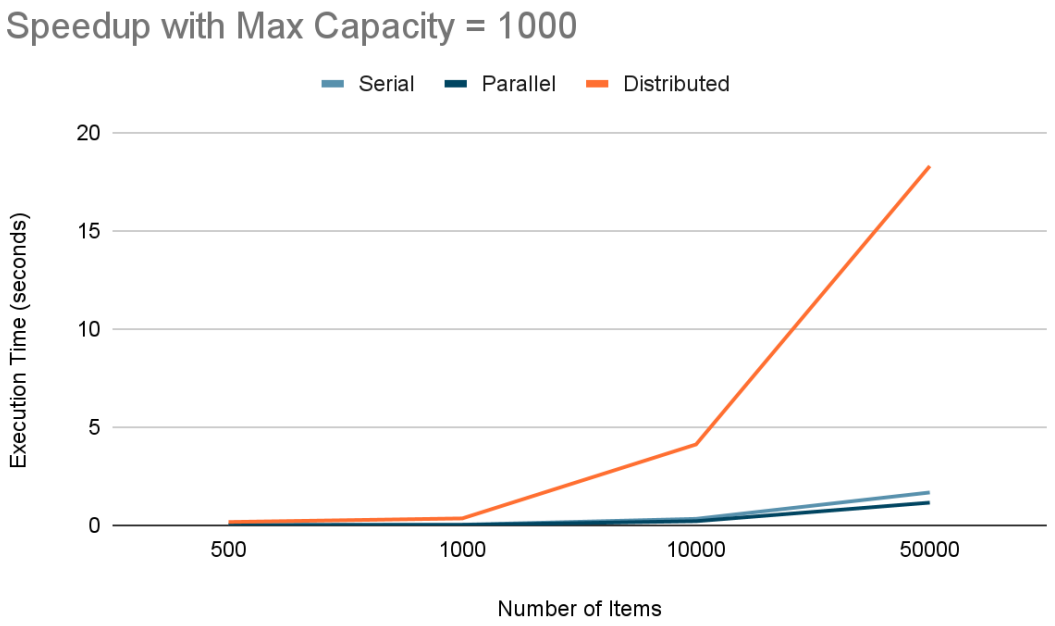
All experiments were run on the CMPT 431 cluster's head node. The number of threads for the parallel version and number of processes for the distributed version was kept constant at 4. We ran all three versions on 4 input sizes ranging from 500 to 50,000 items.

Number of Items	Serial Time	Parallel Time	Distributed Time
500	0.0140698	0.018037	0.038536
1000	0.0209491	0.035424	0.085033
10000	0.169536	0.3426	0.93573
50000	0.818498	1.2402	4.9823

Table 1: Execution times for maximum capacity (S) = 500

Number of Items	Serial Time	Parallel Time	Distributed Time
500	0.0174949	0.011377	0.17083
1000	0.0329211	0.021671	0.35341
10000	0.331472	0.21312	4.1198
50000	1.67374	1.1559	18.29

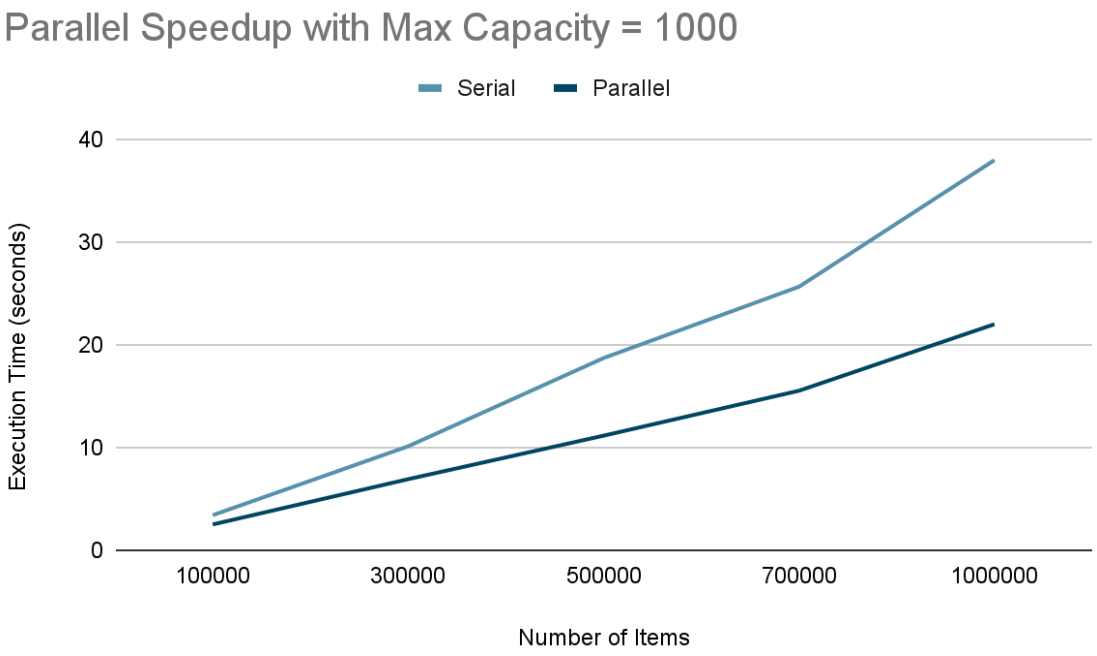
Table 2: Execution times for maximum capacity (S) = 1000



The MPI implementation is significantly slower than the others, but the parallel version shows some speedup from serial. The distributed version limited the upper-bound of input sizes we can run the programs with. To get a better comparison between the speedups of the serial and parallel versions, we conducted experiments on them separately with larger input sizes.

Number of Items	Serial Time	Parallel Time
100000	3.41348	2.5122
300000	10.1154	6.9164
500000	18.6978	11.149
700000	25.6362	15.52
1000000	37.9532	21.991

Table 3: Execution times for maximum capacity (S) = 1000



The speed-up for the parallel program is approximately 0.6x. Asymptotically they grow at the same rate but with lower constants on the parallel side.

Conclusion

For the MPI distributed implementation we should have used more clever task mapping. The first task mapping we implemented used uniform distribution of columns of the dynamic table. We discovered that most of the communication takes place approximately in the middle of the dynamic table; processes that were allocated boundary columns finished their computations much quicker. Then we distributed columns of the dynamic table based on the total number of processes and their ranks. This implementation produces significantly better results compared to the previous, but it still was not fast enough to be comparable with the thread implementation.

We think the unsatisfactory results for the MPI implementation was caused by communication overheads when the program sends boundary rows of the dynamic table. Given more time we could have come up with better mapping of tasks between processes that would minimize communication between rows of the table. We also considered the overhead of sending all the computed table values to the root process so that it can print the indexes of the solution. This communication time was included in the total time that is output by the program. However, it was clear that there was not a significant difference between the single process times and the total time. Therefore, this communication with the root process was not as costly for performance.

We were not able to test the MPI program and measure speedups with larger input sizes because of exponential use of memory for communication. The program gets killed by an OS with input sizes larger than ~50000.

Overall, the Knapsack Problem turned out to be not an easy problem to parallelize, although recent papers[2] show distributed parallel speedup of approximately ~3x.

[2] https://ieeexplore.ieee.org/document/9396489